EXHIBIT G

U 7172156

THE UNITED STATES OF AMERICA

TO ALL TO WHOM THESE PRESENTS SHALL COME:

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office

February 25, 2009

THIS IS TO CERTIFY THAT ANNEXED HERETO IS A TRUE COPY FROM THE RECORDS OF THIS OFFICE OF:

U.S. PATENT: *5,969,705*
ISSUE DATE: *October 19, 1999*

By Authority of the
Under Secretary of Commerce for Intellectual Property
and Director of the United States Patent and Trademark Office

P. SWAIN
Certifying Officer

US005969705A

United States Patent [19]

Fisher et al.

[11] Patent Number: 5,969,705

[45] Date of Patent: Oct. 19, 1999

[54] MESSAGE PROTOCOL FOR CONTROLLING A USER INTERFACE FROM AN INACTIVE APPLICATION PROGRAM

[75] Inventors: Stephen Fisher, Menlo Park; Eric Mathew Trehus, Milpitas, both of Calif.

[73] Assignee: Apple Computer, Inc., Cupertino, Calif.

[21] Appl. No.: 08/816,492

[22] Filed: Mar. 13, 1997

Related U.S. Application Data

[63] Continuation of application No. 08/312,437, Sep. 26, 1994, abandoned, which is a continuation of application No. 08/084,288, Jun. 28, 1993, abandoned.

[51] Int. Cl.[6] G09G 5/00

[52] U.S. Cl. 345/114; 345/345

[58] Field of Search 345/119, 120, 345/118, 113, 114, 115, 343, 344, 345, 346, 347, 348; 395/155, 156, 157, 158

[56] References Cited

U.S. PATENT DOCUMENTS

4,313,113	1/1982	Thornburg .
4,484,302	11/1984	Cason et al. .
4,555,775	11/1985	Pike .
4,688,167	8/1987	Agarwal .
4,698,624	10/1987	Barker et al. .

(List continued on next page.)

OTHER PUBLICATIONS

"Notebook Tabs as Target Location for Drag/Drop Operations",IB, vol. 35, No. 7, Dec. 1992.

Microsoft Corporation, "Microsoft Windows Paint User's Guide," Version 2.0, 1987, pp. 8–10, 44–45.

Microsoft Corporation, "Microsoft Windows Write User's Guide," Version 2.0, 1987, pp. 60–65.

Microsoft Corporation, "Microsoft Word: Using Microsoft Word", Version 5.0, 1989, pp. 69, 88–93.

Screen Dumps from Microsoft Windows V 3.1, Microsoft Corporation 1985–1992 (14 pages).

WordPerfect for Windows V 5.1, WordPerfect Corporation, 1991 (16 pages).

Jeffrey M. Richter, "Implementing Drag–and–Drop," *Windows 3.1: A Developer's Guide*, 2nd Edition, M&T Books, A Division of M&T Publishing, Inc. (1992), pp. 541–577 (Chapter 9).

Charles Petzold, "Windows™ 3.1—Hello to TrueType™, OLE, and Easier DDE; Farewell to Real Mode," *Microsoft Systems Journal*, vol. 6, No. 5 Sep. 1991, pp. 17–26.

Jeffrey Richter, "Drop Everything: How to Make Your Application Accept and Source Drag–and–Drop Files," *Microsoft Systems Journal*, vol. 7, No. 3, May/Jun. 1992, pp. 19–30.

Future Enterprises Inc., A Microcomputer Education Course for: U.S. Department of Commerce "Studen Workbook for Quattro Pro 3.0—Concepts and Basic Uses," 1991 (3 pages).

Inside Macintosh, vol. VI, 1991, pp. 5–1 to 5–117.

Microsoft Windows 3.1, Step by Step, 1991, pp. 168–170.

Apple Computer, Inc., *Inside Macintosh, vol. VI* Table of Contents, 5–1 through 6–117 (1991).

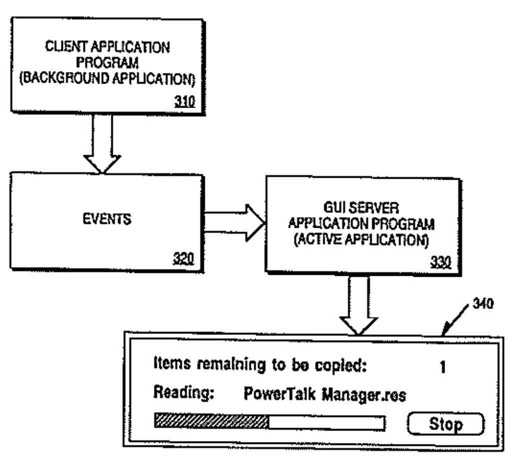
Primary Examiner—Chanh Nguyen

Attorney, Agent, or Firm—Blakely, Sokoloff, Taylor & Zafman

[57] ABSTRACT

Method and apparatus for a first process operative in a computer system controlling a user interface on a computer system display under control of a second process operative in the computer system. An event handler is installed for the second process, the event handler servicing events generated for controlling the user interface display under control of the second process. The first process may then perform a first set of functions in the computer system. The first process generates events for controlling the user interface display, the events related to the functions performed by the first process. The event handler receives the events generated by the first process and updates the user interface on the computer system display according to the events generated by the first process and received by the event handler.

1 Claim, 7 Drawing Sheets



U.S. PATENT DOCUMENTS

4,698,625	10/1987	McCaskill .
4,720,703	1/1988	Schnarel, Jr. et al. .
4,780,883	10/1988	O'Connor et al. .
4,831,556	5/1989	Oono .
4,862,376	8/1989	Ferriter et al. .
4,868,765	9/1989	Diefendorff .
4,905,185	2/1990	Sakai .
4,922,414	5/1990	Holloway et al. .
4,954,967	9/1990	Takahashi .
5,047,930	9/1991	Martens et al. .
5,079,695	1/1992	Dysart et al. .
5,140,677	8/1992	Fleming et al. .
5,157,763	10/1992	Peters et al. .
5,196,838	3/1993	Meier et al. .
5,202,828	4/1993	Vertelney et al. .
5,214,756	5/1993	Franklin et al. .
5,226,117	7/1993	Miklos .
5,226,163	7/1993	Karsh et al. .
5,228,123	7/1993	Heckel .
5,260,697	11/1993	Barrett et al. 345/173
5,287,448	2/1994	Nicol et al. .
5,301,268	4/1994	Takeda .
5,305,435	4/1994	Bronson .
5,333,256	7/1994	Green et al. .
5,339,392	8/1994	Risberg et al. .
5,341,293	8/1994	Vertelney et al. .
5,371,844	12/1994	Andrew et al. .
5,371,851	12/1994	Pieper et al. .
5,400,057	3/1995	Yin .
5,422,993	6/1995	Fleming .
5,442,742	8/1995	Greyson et al. .

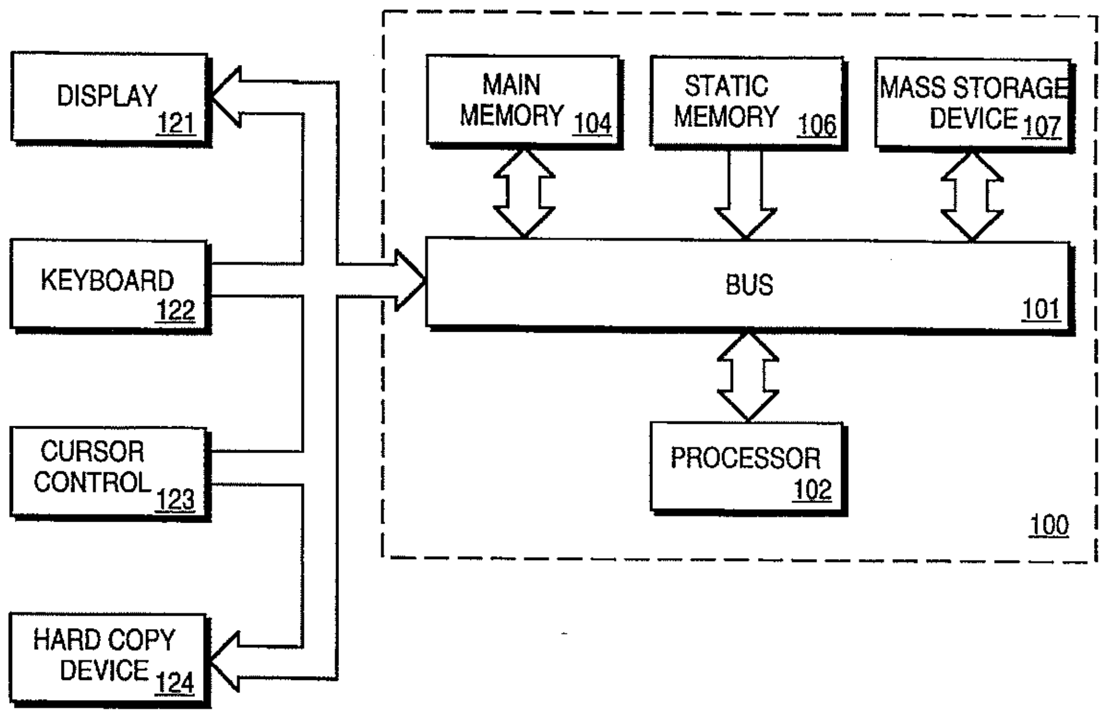

DISPLAY 121
KEYBOARD 122
CURSOR CONTROL 123
HARD COPY DEVICE 124
MAIN MEMORY 104
STATIC MEMORY 106
MASS STORAGE DEVICE 107
BUS 101
PROCESSOR 102
100

FIG. 1

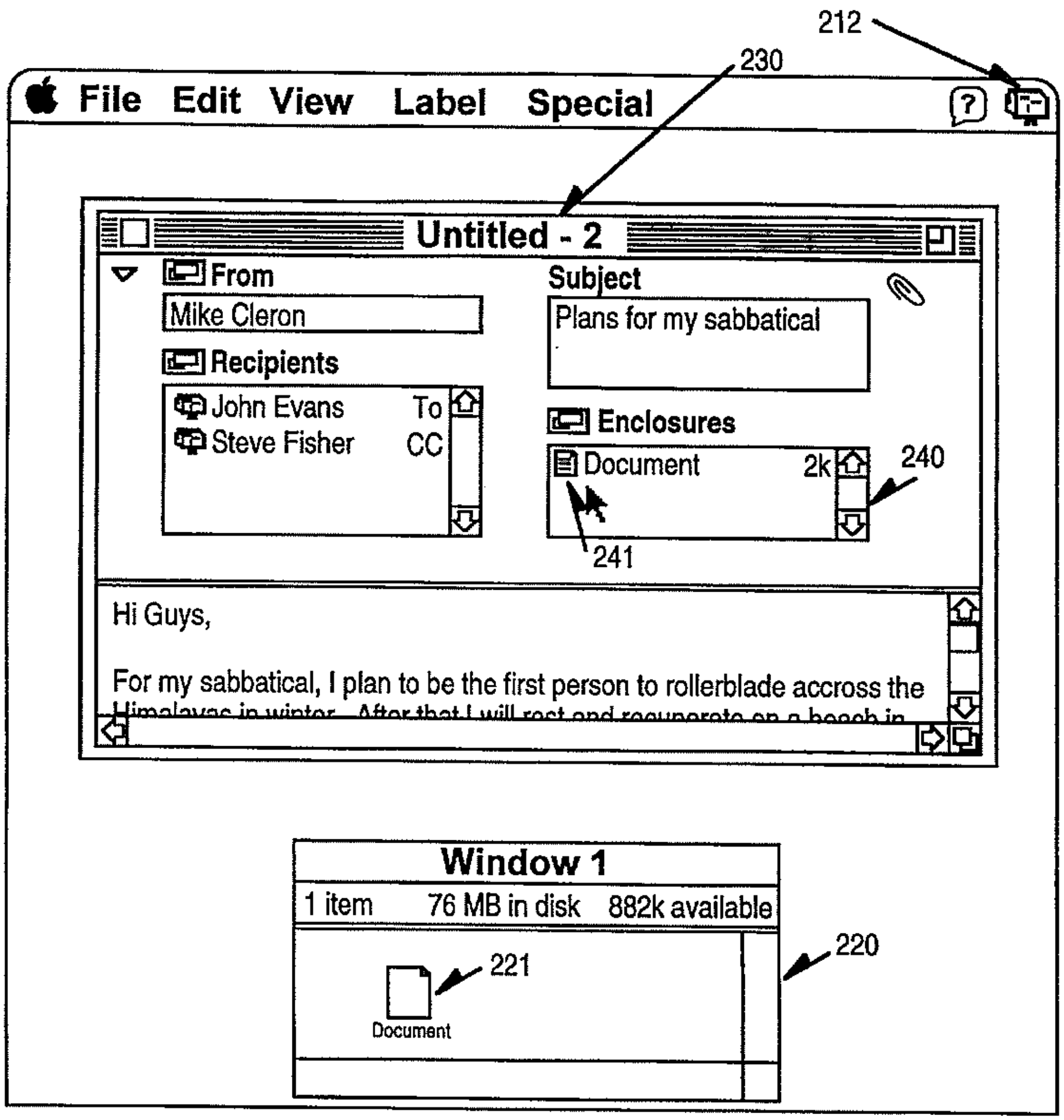


FIG. 2
(PRIOR ART)

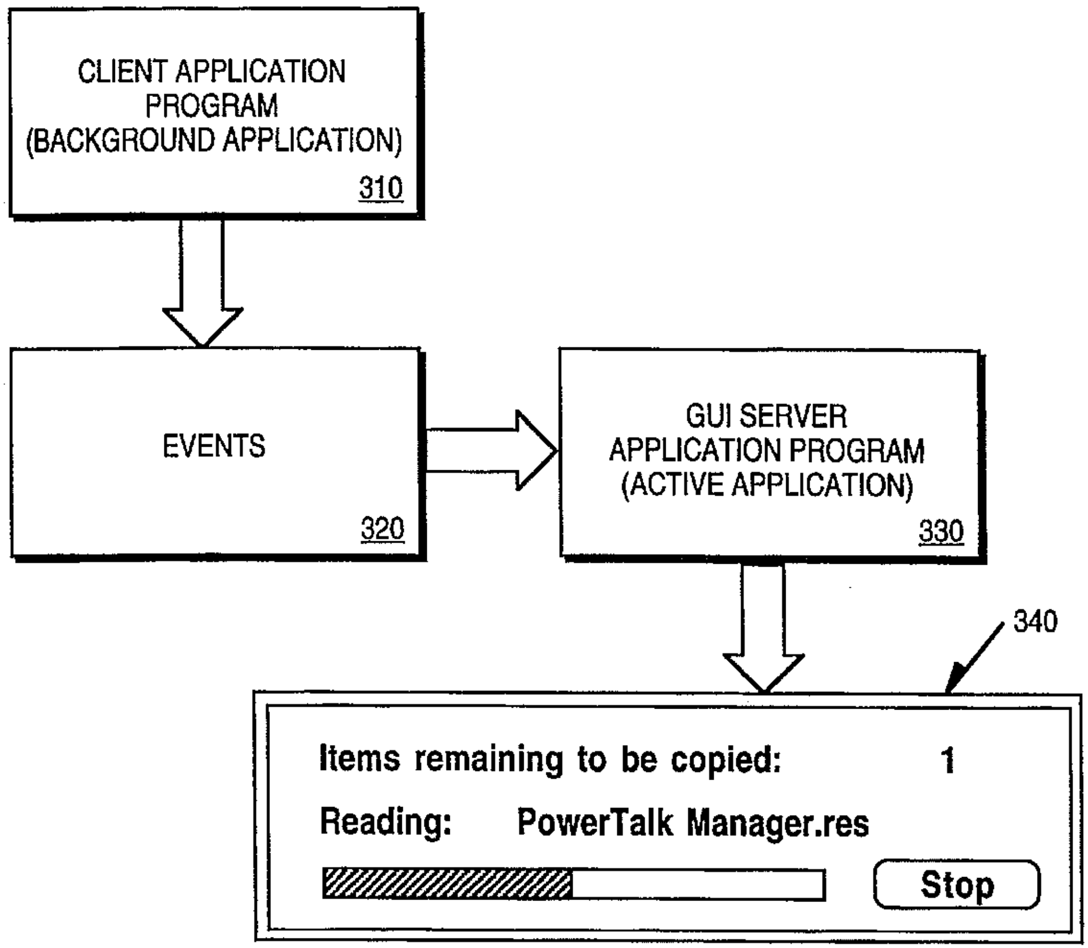

CLIENT APPLICATION PROGRAM (BACKGROUND APPLICATION)
310
EVENTS
320
GUI SERVER APPLICATION PROGRAM (ACTIVE APPLICATION)
330
340
Items remaining to be copied: 1
Reading: PowerTalk Manager.res
Stop

FIG. 3

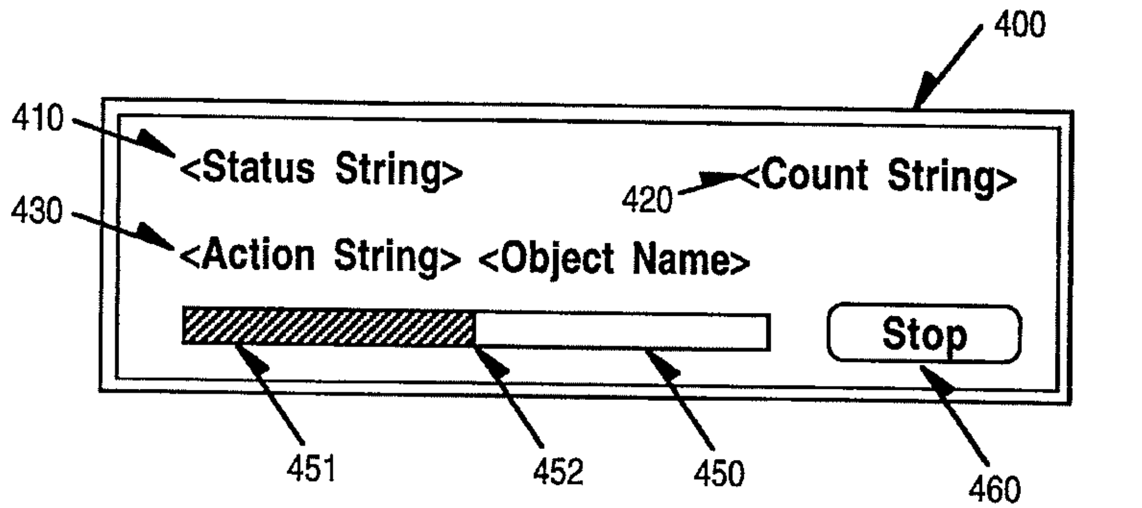


FIG. 4
(PRIOR ART)

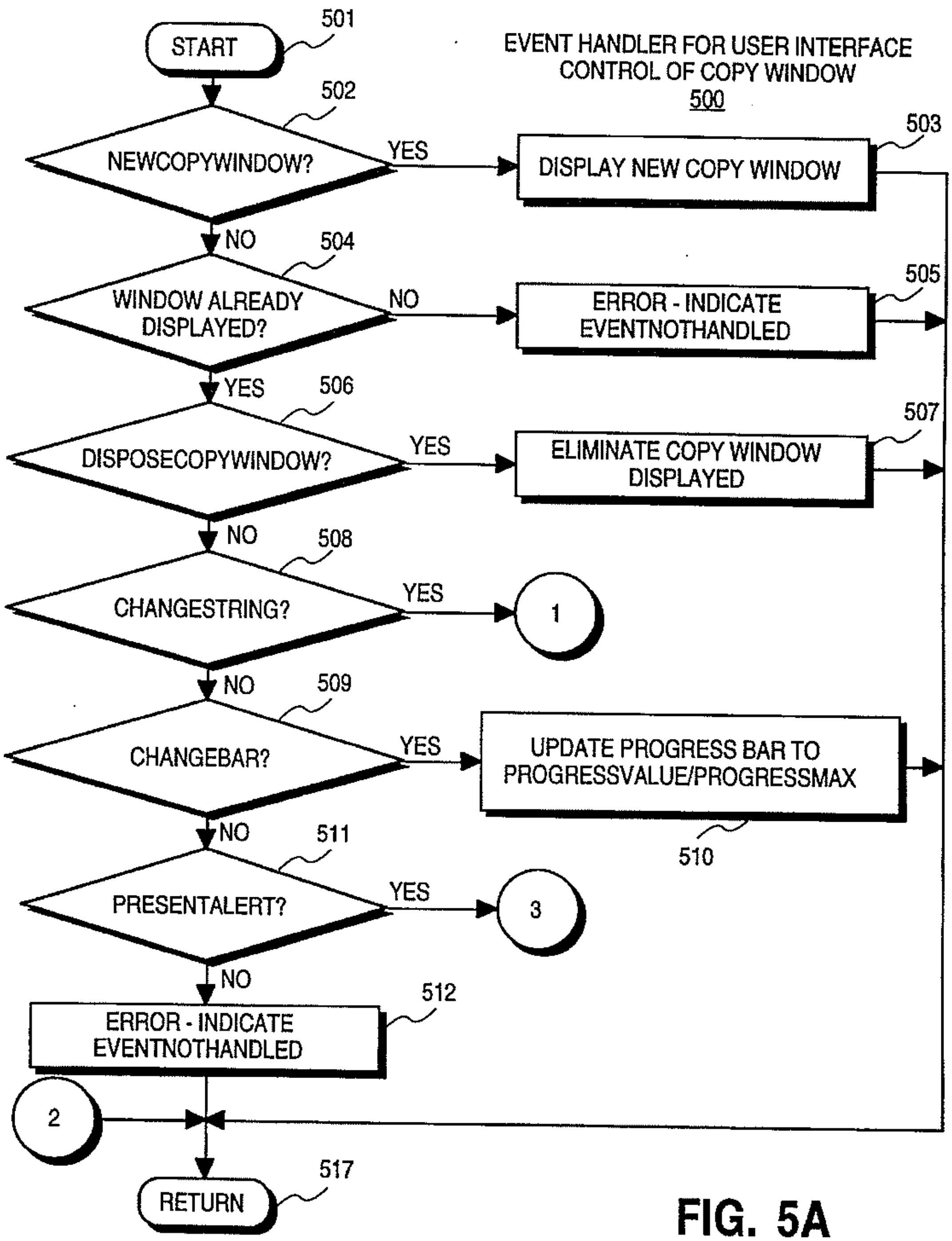

EVENT HANDLER FOR USER INTERFACE CONTROL OF COPY WINDOW
500
START
501
NEWCOPYWINDOW?
502
YES
DISPLAY NEW COPY WINDOW
503
NO
WINDOW ALREADY DISPLAYED?
504
NO
ERROR - INDICATE EVENTNOTHANDLED
505
YES
DISPOSECOPYWINDOW?
506
YES
ELIMINATE COPY WINDOW DISPLAYED
507
NO
CHANGESTRING?
508
YES
1
NO
CHANGEBAR?
509
YES
UPDATE PROGRESS BAR TO PROGRESSVALUE/PROGRESSMAX
510
NO
PRESENTALERT?
511
YES
3
NO
ERROR - INDICATE EVENTNOTHANDLED
512
2
RETURN
517

FIG. 5A

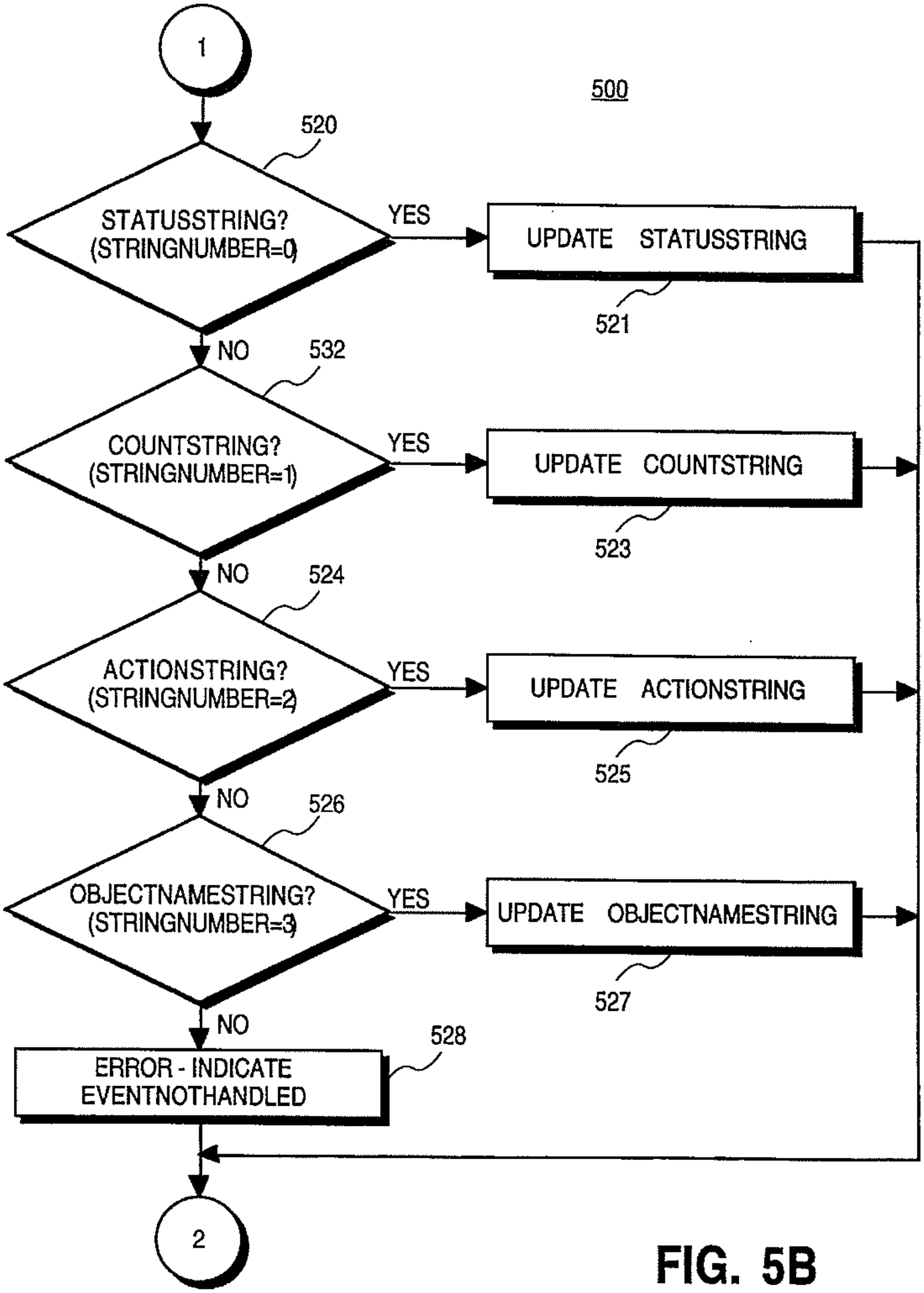

1
500
520
STATUSSTRING?
(STRINGNUMBER=0)
YES
UPDATE STATUSSTRING
521
NO
532
COUNTSTRING?
(STRINGNUMBER=1)
YES
UPDATE COUNTSTRING
523
NO
524
ACTIONSTRING?
(STRINGNUMBER=2)
YES
UPDATE ACTIONSTRING
525
NO
526
OBJECTNAMESTRING?
(STRINGNUMBER=3)
YES
UPDATE OBJECTNAMESTRING
527
NO
528
ERROR - INDICATE EVENTNOTHANDLED
2

FIG. 5B

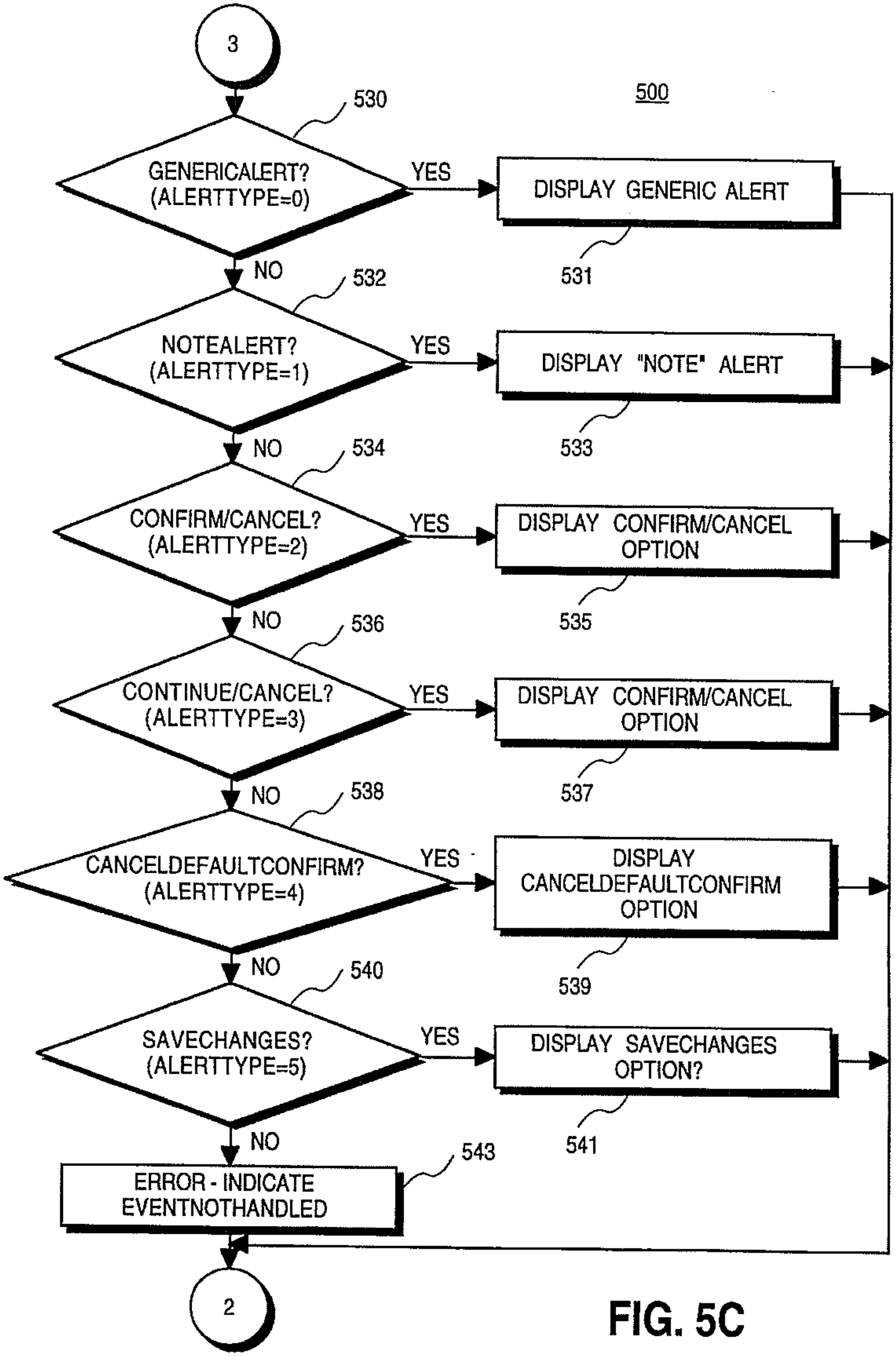

3
500
530
GENERICALERT?
(ALERTTYPE=0)
YES
DISPLAY GENERIC ALERT
531
NO
532
NOTEALERT?
(ALERTTYPE=1)
YES
DISPLAY "NOTE" ALERT
533
NO
534
CONFIRM/CANCEL?
(ALERTTYPE=2)
YES
DISPLAY CONFIRM/CANCEL OPTION
535
NO
536
CONTINUE/CANCEL?
(ALERTTYPE=3)
YES
DISPLAY CONFIRM/CANCEL OPTION
537
NO
538
CANCELDEFAULTCONFIRM?
(ALERTTYPE=4)
YES
DISPLAY CANCELDEFAULTCONFIRM OPTION
539
NO
540
SAVECHANGES?
(ALERTTYPE=5)
YES
DISPLAY SAVECHANGES OPTION?
541
NO
543
ERROR - INDICATE EVENTNOTHANDLED
2

FIG. 5C

MESSAGE PROTOCOL FOR CONTROLLING A USER INTERFACE FROM AN INACTIVE APPLICATION PROGRAM

This is a continuation of application Ser. No. 08/312,437, filed Sep. 26, 1994, now abandoned, which is a continuation of application Ser. No. 08/084,288, filed Jun. 28, 1993 status: abandoned.

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to user interface control in a computer system. More specifically, the present invention relates to a messaging protocol which allows one application program to specify the appearance of an interface of a computer system while under control of a second application program.

2. Background Information

In multitasking operating systems, such as the Macintosh brand System 7.0 operating system available from Apple Computer, Inc. of Cupertino, Calif., typically, only one application program is given complete control of the user interface in order to prevent conflicts. There are circumstances, however, in which an application program which does not currently have control of the user interface will require that some information be presented to a user of the computer system. Typically, in the prior art, in these situations, the "inactive" application program must be "brought to the front" or made the active application (one in control of the user interface) in order for user interface control to become available to the application program. If events or other activities occur within the process that does not have control of the user interface, then the user may not be informed of the activity until after the activity has taken place, when the user brings the background application to the front. There are some circumstances in which the delay between the occurrence of the action within the background process, and the failure to provide feedback upon the computer system display may pose a substantial problem. For example, data may be overwritten, the user may wish to abort the task being performed, or he may wish to take corrective measures to otherwise address the activity occurring in the background task. There thus has arisen a need for background process to control the user interface which is currently under control of an "active" or foreground process within a computer system.

Another situation which frequently occurs is when one application program requires a complex service such as a file copying mechanism, but yet does not possess the necessary code in order to perform these tasks. An active application can use the services of the inactive application's processes without possessing the necessary code, and the inactive application program may drive the user interface of the "active" application program in order to provide feedback that the complex operation is taking place. Unfortunately, prior art techniques have no mechanism for allowing this to take place.

SUMMARY AND OBJECTS OF THE PRESENT INVENTION

One of the objects of the present invention is to allow a background application to provide user interface feedback when it is not the currently active application program.

Another of the objects of the present invention is to provide a protocol wherein a background application program may direct a foreground application program to control its user interface in a specified way.

Another of the objects of the present invention is to allow a background application program to communicate with a foreground application program for controlling the user interface of a computer system display.

Another object of the present invention is to allow a foreground application program controlling a user interface to take advantage of the services of a background application program.

Method and apparatus for a first process operative in a computer system controlling a user interface on a computer system display under control of a second process operative in the computer system. An event handler is installed for the second process, the event handler servicing events generated for controlling the user interface display under control of the second process. The first process may then perform a first set of functions in the computer system, in one embodiment, such as file management functions (e.g. copying and/or moving of files in the file system). The first process generates a first set of events for controlling the user interface display, the first set of events related to the first set of functions performed by the first process. For example, in various embodiments, feedback may be given about the progress of the file management functions (such as copying/moving specific files, reading from a source, and copying to a destination). The event handler receives the first set of events generated by the first process and updates the user interface on the computer system display according to the events generated by the first process and received by the event handler. This may include, showing the progress of the file management operation, and alerting the user of any abnormal conditions.

Other features, objects, and advantages of the present will become apparent from viewing the figures and the description below.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention is illustrated by way of example and not limitation in the figures of the accompanying in which like references indicate like elements and in which:

FIG. 1 shows an example of a computer system architecture upon which one embodiment of the present invention may be implemented.

FIG. 2 shows a situation in which a file system manipulation may be performed when the filed management task is not the "active" application program (the program controlling the user interface).

FIG. 3 shows an event-driven architecture which is used in one embodiment of the present invention for allowing a background task to control a user interface of a foreground application program.

FIG. 4 shows an example of user interface display which may be controlled directly by a foreground application program or "server" process, but which may be directed by and whose functionality may be provided for by a background process (or "client").

FIGS. 5*a*–5*c* show process flow diagrams of an event handler which is registered for use by a server application program to service events generated by a client application program for user interface control.

DETAILED DESCRIPTION

The present invention relates to a messaging protocol between processes and a computer system wherein a first

process (e.g., a client process) sends messages to a second process (e.g., a server process) so that the client process can direct the appearance of the user interface under control of the server process. In this manner, the client process performs certain functions, and the server process controls all user interface functions such as the display of feedback for those functions. For the remainder of this application, various process steps, apparatus, data structures, message formats, events, parameters, and other information will be discussed in detail, however, these are merely for illustrative purposes and are not intended to limit the present invention. It can be appreciated by one skilled in the art that many departures and modifications may be made from these specific embodiments without departing from the overall spirit and scope of the present invention.

Referring to FIG. 1, a system upon which one embodiment of the present invention is implemented is shown as **100**. **100** comprises a bus or other communication means **101** for communicating information, and a processing means **102** coupled with bus **101** for processing information. System **100** further comprises a random access memory (RAM) or other volatile storage device **104** (referred to as main memory), coupled to bus **101** for storing information and instructions to be executed by processor **102**. Main memory **104** also may be used for storing temporary variables or other intermediate information during execution of instructions by processor **102**. Computer system **100** also comprises a read only memory (ROM) and/or other static storage device **106** coupled to bus **101** for storing static information and instructions for processor **102**, and a data storage device **107** such as a magnetic disk or optical disk and its corresponding disk drive. Data storage device **107** is coupled to bus **101** for storing information and instructions. Computer system **100** may further be coupled to a display device **121**, such as a cathode ray tube (CRT) or liquid crystal display (LCD) coupled to bus **101** for displaying information to a computer user. An alphanumeric input device **122**, including alphanumeric and other keys, may also be coupled to bus **101** for communicating information and command selections to processor **102**. An additional user input device is cursor control **123**, such as a mouse, a trackball, stylus, or cursor direction keys, coupled to bus **101** for communicating direction information and command selections to processor **102**, and for controlling cursor movement on display **121**. Another device which may be coupled to bus **101** is hard copy device **124** which may be used for printing instructions, data, or other information on a medium such as paper, film, or similar types of media. Note, also, that any or all of the components of system **100** and associated hardware may be used in various embodiments, however, it can be appreciated that any configuration of the system may be used for various purposes as the user requires.

In one embodiment, system **100** is one of the Macintosh® family of personal computers such as the Macintosh® Quadra™ or Macintosh® Performa™ brand personal computers manufactured by Apple® Computer, Inc. of Cupertino, Calif. (Apple, Macintosh, Quadra, and Performa are trademarks of Apple Computer, Inc.). Processor **102** may be one of the 68000 family of microprocessors, such as the 68030 or 68040 manufactured by Motorola, Inc. of Schaumburg, Ill.

Note that the following discussion of various embodiments discussed herein will refer specifically to a series of routines which are generated in a high-level programming language (e.g., the C++ language available from Symantec of Cupertino, Calif.) and compiled, linked, and then run as object code in system **100** during run time. It can be

appreciated by one skilled in the art, however, that the following methods and apparatus may be implemented in special purpose hardware devices, such as discrete logic devices, large scale integrated circuits (LSI's), application-specific integrated circuits (ASIC's), or other specialized hardware. The description here has equal application to apparatus having similar function.

Graphical User Interface

Before discussing the preferred embodiment in detail, a brief overview of the user interface used in this system is required. A "windowing" or graphical user interface (GUI) operating environment is used wherein selections are performed using a cursor control device such as **123** shown in FIG. 1. Typically, an item is "selected" on a computer system display such as **121** using cursor control device **123** by positioning a cursor, or other indicator, on the screen over (or in proximity to) an object on the screen and by depressing a "selection" button which is typically mounted on or near the cursor control device. The object on the screen is often an icon which has an associated file or operation which the user desires to use in some manner. In order to launch a user application program, in some circumstances, the user merely selects an area on a computer display represented as an icon by "double clicking" the area on the screen. A "double click" selection is an operation comprising, while positioning the cursor over the desired object (e.g., an icon), two rapid activations of the selection device by the user. "Pull-down" or "pop-up" menus are also used in the preferred embodiment. A pull-down or pop-up menu is a selection which is accessible by depressing the selection button when the cursor is pointing at a location on a screen such as a menu bar (typically at the top of the display), and "dragging" (moving cursor control device **123** while the selection button is depressed) until the selection the user wishes to access is reached on the pull-down menu. An item is indicated as being "selected" on a pull-down menu when the item is highlighted or displayed in "reverse video" (white text on a black background). The selection is performed by the user releasing the selection device when the selection he wishes to make is highlighted. Also, in some GUI's, as is described in the background above, the "selection" and "dragging" of items is provided to move files about in the file system or perform other system functions. These techniques include "dragging and dropping" which comprises making a "selection" of an icon at a first location, "dragging" that item across the display to a second location, and "dropping" (e.g., releasing the selection device) the item at the second location. This may cause the movement of a file to a subdirectory represented by the second location.

Note also that GUI's may incorporate other selection devices, such as a stylus or "pen" which may be interactive with a display. Thus, a user may "select" regions (e.g., an icon) of the GUI on the display by touching the stylus against the display. In this instance, such displays may be touch or light-sensitive to detect where and when the selection occurs. Such devices may thus detect screen position and the selection as a single operation instead of the "point (i.e., position) and click (e.g., depress button)," as in a system incorporating a mouse or trackball. Such a system may also lack a keyboard such as **122** wherein the input of text is provided via the stylus as a writing instrument (like a pen) and the user handwritten text is interpreted using handwriting recognition techniques. These types of systems may also benefit from the improved manipulation and user feedback described herein.

One problem solved by the present invention is illustrated with reference to FIG. 2. Window **200** of FIG. 2 illustrates

a typical Macintosh user interface display while one application program, such as an electronic mail program, controls the user interface display. This is illustrated by the icon present in the upper right-hand portion **212** of the display **200**. The operating system only allows a single application program to control the user interface at any given time. However, other application programs such as, those performing file and/or program management functions (e.g., the "Finder" within the Macintosh brand operating system), allow the launching of applications, programs, and the movement of files within the file system. In one embodiment of the present invention, a user may decide to "enclose" a file represented by icon **221** in the file system with the mail message represented on window **230**. The application program controlling window **230** does not possess file transfer or file transfer feedback capabilities, whereas the File System Manager (known as the "Finder" in the Macintosh) does possess these capabilities. Therefore, it is desired that the application program controlling window **230** have certain functions and user interface capabilities of the Finder. In a typical prior art systems, feedback is provided by the file management function to show that the file movement or copy operation is taking place. This typically takes the form of a progress bar on a typical prior art user interface to show the progression of the file transfer operation as it takes place in the file system. For example, if a plurality of files are moved in the file system, or copied from one media device to another, file names showing each transfer of each file, and a darkened representation is shown in the progress bar to show overall completion of the copying of the files is represented on the progress bar. This will be illustrated in more detail below.

Event-Driven Architecture

An operation such as copying or moving files from one location to another in the file system is a nontrivial task. In this embodiment, the application program controlling the user interface defers to the background task the "Finder" so that it may perform the file management functions for transfer and/or copy of the file(s). For example, several tasks need to be performed by the copy/movement process prior to copying or moving the files. For example, the destination subdirectory or other file location needs to be scanned to determine whether any of the transferred file names are equivalent to those already in the subdirectory. If so, the user needs to be alerted in order to determine whether he wishes to overwrite the existing files at that location or, perhaps, use a different name. In another instance, there may not be sufficient space at the destination to which the files are being moved for writing the files. In this instance, the user is alerted that there was not sufficient space on the storage medium to store the files, is informed that the operation was not successful, and any file(s) already written or other intermediate file information can be deleted. However, in a situation such as that illustrated in FIG. **2**, the underlying file management process cannot present this user feedback or alert information to the user because it does not presently have control of the user interface. The process having control of window **230**, an electronic mail application program, is currently in control of the user interface. Thus, an improved means for allowing the background process (e.g., the Finder) to control the user interface is used in various embodiments of this invention to present feedback to the user regarding the underlying functions that are taking place. This is performed via interprocess communication, in the Macintosh brand operating system, using Events and the accompanying operating system Event Manager and Apple

Event Manager available from Apple Computer of Cupertino, Calif.

Interprocess communication is an important aspect of modern computer system design. For example, such interapplication communication has provided in modern computer systems, such as the Macintosh brand computer's operating System 7.0, available from Apple Computer of Cupertino Calif., through a mechanism known as the Event Manager. The Event Manager and the Apple Event Manager are used for handling a wide variety of functions within application programs such as detecting user actions—key clicks, selections using a cursor control device, the detection of the insertion of disks into a disk drive on the computer system, opening files, closing files, or other actions within the computer system. Typically, processes running within a Macintosh brand computer system comprise a main program loop known as an "event" loop which detect the occurrence of these events in the system. Then, the application typically branches to portions of the program to allow the event to be serviced. Such an event driven architecture forms the core of many application programs within many different types of computers and operating systems in present use but, in this embodiment, resides in a system such as **100** described above.

Various embodiments of the present invention use the Event Manager and the Apple Event Manager supplied by Apple Computer for interprocess communication between applications programs which are operative within computer system **100** during run time to implement the features described herein. The event driven architecture for message passing between a first application program (e.g., a client application program which is operative in the background), and a second application program (e.g., a server application program which is the active application controlling the user interface), is illustrated with reference to FIG. **3**. For example, using the event driven architecture specified in *Inside Macintosh, Volume* 6, pages 5–1 through 6–118, "client" application program **310** communicates with the Graphical User Interface (GUI) "server" application program or active application program **330** via events **320**. Each of these events are generated by the client application program **310** and are detected by the Apple Event Manager. GUI server application **330** registers a process with the Apple Event Manager known as an Event Handler so that whenever defined events are detected, the Apple Event Manager forwards the event(s) to the registered handler(s) and cause the handler(s) to be invoked and service the events. At that point, the handler may determine the appropriate action to take place. In various embodiments of the present invention described herein, the registered event handler for GUI server **330** will cause the activation and modification of a user interface display, in this case, copy window **340** illustrated in FIG. **3**. Upon the launching of the GUI server application program **330**, or any application program which may become an active application program during specified user actions (e.g., the copying of file(s), the server application program will register with the Apple Event Manager the handler(s) which are used to service the events, including those generated by any potential client application programs (e.g., **310** of FIG. **3**). In the situation where a defined event is not serviced by any handler which are registered by the application program, then, a default handler supplied by the operating system is instead used for servicing the events.

For the remainder of this application, in the embodiment discussed herein, it will be assumed that the "server" (e.g., **330** of FIG. **3**) has registered a handler which will service

user interface events. It will also be assumed that a client program (e.g., **310** of FIG. **3**) provides the underlying functionality for performing the actions represented by the user interface (e.g., copying files), which occurs upon the inactive program detecting that a file should be copied (or moved) from one directory to another, such as a directory for "Enclosures" within an electronic mail application program. This function may also be requested by server process **330** sending an event to a handler registered for client process **310**, such as "CopyFile" 'File 1' to 'Enclosures.'" The mechanics of this operation, however, will not be described in detail because they are beyond the scope of the present invention.

Client to Server Events for Controlling the User Interface

The following events are defined in this new protocol for communicating from client **310** to user interface server **330**:

1. NewCopyWindow;
2. DisposeCopyWindow;
3. ChangeString;
4. ChangeBar; and
5. PresentAlert.

As client application program **310**'s file copy operations progress, certain of these events are issued by client process **310** to server process **330**. Moreover, communication is provided via another set of events from server **330** to client **310** to indicate the success of the action indicated by the events, and user interface feedback in response to information presented on the user interface by server **330**. A description of each of the specific events and the parameters used in each of these events will now be discussed with reference to FIG. **4**.

400 of FIG. **4** illustrates a typical copy window which is displayed during a file copying operation well known in the prior art. However, each of the informative portions of the window are displayed with corresponding parameter name for the event in angled brackets (e.g., <status string> **410**, <action string> **430**, <count string> **420**, and <object name> **440**), which is replaced by the strings specified within parameters associated with the event(s) issued by client process **310**. The events and parameters associated with the events will now be discussed.

NewCopyWindow

The NewCopyWindow event is signaled by client application program **310** to indicate that a new copy window (e.g., **400** in FIG. **4**) should be displayed. Using typical prior art user interface commands, the server application program's handler creates upon the display screen a copy window **400** as is illustrated in FIG. **4** with the appropriate strings specified in the event parameters. Each of the event parameters for the NewCopyWindow are specified in the following order:

1. actionString (e.g., "reading"/"writing"/"verifying")
2. objectName (name of object being copied)
3. statusString (e.g., "Preparing To Copy," "Items remaining," etc.)
4. countString (how many items are being copied)
5. progressValue (an initial value, usually 0)
6. progressMax (a maximum value)

actionString is used for specifying the operation being performed. In typical prior art copy operations, the value thus is one of the following three strings: "Reading"; "Writing"; or "Verifying", for specifying the operation being performed. The specified action string is placed into region **430** of the copy window **400** upon detection of the NewCopyWindow event.

objectName is used for specifying the string which will appear at region **440** on copy window **400**. It is used for specifying the name of the object or file being copied. Feedback can thus provide to the user which object is currently in the process of being read, written, or verified.

statusString **410** is used to specify the intermediate status of the copy operation taking place. For example, in certain prior art systems, this string may read "Preparing to Copy," "Items Remaining," etc. The status string indicates to the user the current status of the copy operation taking place.

countString is used for specifying the value which is shown at region **420** of display **400**, such as the number of items (e.g., files or bytes) which are being copied. Thus, in one situation, this string may contain "120 bytes" when a file or file(s) of 120 bytes in length are being copied.

progressvalue and progressMax are used for specifying the status of progress bar **450**. For example, the progressMax parameter is used for specifying some maximum value at which the progress bar will be completely darken. In an instance where a total of 120 bytes are being copied, progressMax may be equal to an integer value, such as 120. The progressValue parameter will thus be used to specify an intermediate value from some initial value (in one embodiment, the integer 0) to the progressMax value. Thus, on the display, progress bar **450** may be filled with a darkened region **451** up to an intermediate position **452** based upon the fraction progressValue/progressMax. For example, if progressValue equals 60 and progressMax equals 120, then progress bar **450** will have a representation such as that shown in FIG. **4** wherein $^{60}/_{120}$ or ½ of the progress bar has been darkened. Feedback is thus provided to the user to illustrate the current completion of the copy operation. In typical situations, the progressvalue will have an initial value such as 0, and the progressMax value will be some nonzero value, for example, equivalent to an integer representing the maximum number of bytes to be copied.

DisposeCopyWindow

The DisposeCopyWindow event is used for indicating the termination of a copy operation. This event causes the server's handler to remove window **400** from the display using well known prior art interface techniques. The event has no parameters because it merely removes from the display the currently displayed progress bar window.

ChangeString

The ChangeString event has the following parameters:

stringNumber
statusString = 0
countString = 1
actionString = 2
objectNameString = 3
stringValue

stringNumber—For each of the above specified values of stringNumber, the identified string modified in copy window **400** using the ChangeString event according to the string contained in stringValue. statusString **410**, countString **420**, actionString **430**, or objectNameString **440** may be modified within, copy window **400** illustrated in FIG. **4**. The client application may thus cause updates to be performed within copy window **400** so that the user is informed of the current status of the copy operation (such as a current file being copied)

stringValue is a string which will replace the string specified by the integer value contained in stringNumber.

ChangeBar

The ChangeBar event has the following parameters:

progressValue (a current value)

progressMax (a maximum value) Each of these parameters are integer values specifying the current progression and the maximum progression of progress bar **450** of copy window **400**, is discussed with reference to the NewCopyWindow event above. The progress bar is thus adjusted to have a filled in representation, such as that shown as **451** according to the fraction progressValue/progressMax. The progress bar is update by the server's handler using standard prior art user interface commands.

PresentAlert

The following parameters are defined for the PresentAlert event:

alertType	
	GenericAlert = 0
	NoteAlert = 1
	Confirm/Cancel = 2
	Continue/Cancel = 3
	CancelDefaultConfirm = 4
	SaveChanges = 5
alertString	

alertType is an integer value which is used for specifying the type of alert displayed which is displayed to the user. Note that these alerts are all similar to those which are displayed in typical prior art copy operations upon detection of certain conditions, abnormal or otherwise. Some of these specified alerts require that the user respond. For example, the Confirm/Cancel alert displays a window which requests that the user "confirm" or "cancel" an ongoing operation. For example, the Confirm/Cancel alert may be used when the same file name is detected at a destination directory for a file which is being copied. Any of the standard alert windows which may be used in certain prior art copy operations may be specified using the proper alertType integer value. User responses to alerts will be provided with events from user interface server **330** to client **310** via another set of events discussed below.

alertString is a string value indicating the associated message to be associated with the alert. For example, the client application program may determine that a file name having an equivalent file name to a file being copied already resides at the destination application. In this event, the alertString may contain a message such as "File 'My File' already exists. Replace?" In any event, using the foregoing alert parameters, the PresentAlert event may specify appropriate alerts to the user.

Server to Client Events

All of the above events are shown for illustrative purposes only and are for the client application process **310** (e.g., the "Finder" performing the copy operation) alerting server process **330** (e.g., an electronic mail application program) to change the user interface display in a specified manner. However, other events may also be defined to specify other changes to the user interface display and for other operations, especially in instances where user interface response(s) to alerts are required. Because the background process (e.g., client process **310**) cannot detect user interface actions (such as selections on the display), the handler for server process **330** must detect these actions and transmit response event messages to client process **310**. In this case, client **310** will have its own event handler registered for servicing events issued by server **330** to client **310**. For example, if the user wishes to "cancel" an operation, an event entitled "CopyCancel" may be issued to the client process **310** in one embodiment of the present invention so that any ongoing operation(s) may be aborted. This is detected by a user selecting "Stop" button **460** at any time during the operation. The cancel operation may be detected by server **330** by the detection of a "mouseDown" event at a specific location, such as "Stop" button **460**, or its keyboard equivalent (e.g., a command period combination in the Macintosh). In this case, client application **310**'s handler may be alerted that an abort was indicated and take appropriate action.

In another embodiment, responses to alerts such as file overwrite messages may be sent from server **330** to client **310**. In this instance, an integer may be passed as a parameter wherein one value of the integer (e.g., 0) causes the operation to be aborted or a second value (e.g., 1) causes the file overwrite to be confirmed. Responses to alerts are performed using other defined events from GUI server **330** to client **310**. The user may be presented with the option of either confirming replacement of the file or canceling the copy operation. In one embodiment, when an alert is presented, client **310** remains idle until a response is made by a user on the user interface display. Then a corresponding response event (e.g., AlertReply) with a response parameter (e.g., an integer value Result containing an integer 0 indicating confirmation of the operation or integer 1 indicating canceling of the operation) is generated by server **330** to client **310** to either confirm or cancel the operation being performed by client **310**.

Other user responses to queries, such as alerts, errors, or other conditions, may also be responded to in this manner, as detected by GUI server **330** and sent to client **310** via a registered handler.

Event Handling by Server Application Program

FIGS. *5a–5g* show a process flow diagram of a typical event handler which may be registered by server application program **330** and service events generated by a background process for controlling the user interface using the messaging protocol of one embodiment of the present invention. For example, such an event handler may be registered using the Apple Event Manager described in *Inside Macintosh, Volume VI*, Chapter 6, wherein all of the above-described events are handled by this single handler **500**. Handler **500** will typically have a process entry point, such as **501** illustrated in FIG. *5a*, and comprise an IF or CASE programming statement in a typical high level programming language or other condition checking loop, which is illustrated in the remainder of the figures. Then, each of the events may be checked for, and upon detection of a specific event, the user interface display specified by the event and associated parameters may be displayed upon system display **121**. For example, it will be determined at step **502** whether a NewCopyWindow event has been detected. If so, then a new copy window (e.g., **400**) is displayed at step **503** with the specified parameters, such as statusString **410**, countString **420**, actionString **430**, objectName **440** , and having the progress bar **450** with an appropriate representation as defined by the progressValue/progressMax parameters passed within the event. Upon display of the new copy window **400** at step **503**, process **500** continues and returns at step **517**.

If, however, a NewCopyWindow event was not detected at step **502**, and a window is not currently displayed as

detected at step **504**, then an error condition may be indicated at step **505**, such as by issuing an event from server **330** to client **310** to specify an error (e.g., "EventNotHandled") to specify that the event was not serviced. Then, event handler **500** may exit at step **517**. If, however, a window has already been displayed, then the condition for the various remaining events defined in the messaging protocol may be checked for using a suitable programming construct such as a CASE statement or other similar condition-checking statement(s). For example, it is determined at step **506** whether the DisposeCopyWindow event has been detected. If so, then the copy window displayed upon computer system display **121** is eliminated at step **507** using well-known prior art user interface operations, and handler **500** returns at step **517**.

Upon the detection of a ChangeString event, as detected at step **508**, then process **500** proceeds to a more detailed sequence of steps to determine the value passed within the string number, as reflected on Figure *5b*. As is illustrated in FIG. *5b*, it is determined using a condition checking loop, such as a CASE statement or other programming construct, what the value of stringNumber is. For example, at step **520**, it is determined whether stringNumber indicates that the statusString should be modified. If so, then statusString **410** on display **400** is updated at step **521**, and the handler returns at step **517**. If, however, the countString is specified at step **522** (when stringNumber=1), then the count string (e.g., **420**) is updated at step **523**, and the handler returns at step **517**. If, however, the actionString should be modified (stringNumber=2), as detected at step **524**, then it is updated on the copy window. If, however, stringNumber=3 indicating that objectNameString **440** is sought to be updated, as detected at step **526**, then objectNameString **440** is updated at step **527**, and handler **500** returns at step **517** of FIG *5a* . Any other string number results in an error being generated at step **528**, and a return from the handler at step **517** with an appropriate error event message to client **310**, such as "EventNotHandled," indicating that the handler did not service the event.

Process **500** of FIG. *5a* proceeds to step **509** if a ChangeString event was not detected. Step **509** determines whether the ChangeBar event has been detected. If so, then handler **500** proceeds to step **510** which updates progress bar **450** using the progressvalue and progressMax parameters passed in the event. If the value(s) passed are invalid, an error may be indicated via a response event and the update to the progress bar abort.

If, however, the ChangeBar event is not detected at step **509**, then the handler proceeds to determine whether a PresentAlert event has been detected at step **511**. Various types of alerts can then be checked for, as illustrated in FIG *5c*, by checking the alertType parameter. For example, each of the steps illustrated at steps **530**–**540** on FIG. *5c* may be conformed using a typical high-level programming construct such as a CASE statement. Then, upon detection of the corresponding value in the alertType parameter, the associated alert is displayed. For instance, for alertType=0, as detected at step **530**, the generic alert is tested for and then displayed at step **531**. At step **532**, the NoteAlert is tested for (with alertType=1) and displayed at step **533**. At step **534**, the Confirm/Cancel alert is tested for (with the alertType=2), and it is displayed if detected at step **535**. At step **536**, if the Continue/Cancel alert type is detected (alertType=3), then the Confirm/Cancel option window is displayed at step **537**. At step **538**, the CancelDef aultConfirm option is tested for (alertType=4), and the corresponding option is displayed at step **539**. Finally, the SaveChanges parameter is tested for (alertType=5) at step **540** and then displayed at step **541**. If any other alertType value is detected, an error is indicated at step **543**, and the process returns at step **517** of FIG. *5a*. Otherwise, upon completion of detection of any of the above alertType values, then the option previously displayed is saved for use when the next event is detected in the event handler at step **542**, and process **500** returns at step **517** of FIG. *5b*.

At any rate, upon detection of the all the previous events, if the events are serviced, then an event message from server **330** to client **310** such as EventHandled is issued, and handler **500** returns at step **517**. Otherwise, any events detected which do not fall into one of the categories tested for or other events which are not serviced may issue a suitable error event message, such as EventNotHandled at step **517** to indicate to client **310** that the event was not serviced. Then, the client may take appropriate actions via its own event handler.

Thus, an invention for a background application controlling the user interface of a foreground application has been described. Although the present invention has been described particularly with reference to specific embodiments as illustrated in FIGS. **1**–*5c*, it may be appreciated by one skilled in the art that many departures and modifications may be made by one of ordinary skill in the art without departing from the general spirit and scope of the present invention.

What is claimed is:

1. In a computer system comprising a processor, a display, a memory, a user input device, a first process operative in the computer system, a second process operative in the computer system as a foreground process and a user interface on said computer system display under the control of the second process, a method for the first process to perform operations for the second process and control a content of the user interface on said computer system display, said content under control of the foreground second process operative in said computer system, said first process controlling the content to display information regarding the operations performed by the first process for the second process, said method comprising the following steps:

a. installing an event handling process as part of said second process, said event handling process when said second process is operative in said computer system, servicing events generated by the first process for controlling said user interface display under control of said second process;

b. said second process initiating said first process to perform operations for said second process, said second process operative in the foreground and said first process operative in the background;

d. said first process generating events for controlling said user interface display while the second process remains as a foreground process and the first process is a background process, said events providing information regarding the operations performed by said first process for the second process; and

e. said event handling process receiving events generated by said first process, said event handling process updating said user interface on said computer system display according to said events generated by said first process, while said first process remains in the background, and received by said event handling process.

* * * * *

EXHIBIT H

US005946647A

United States Patent [19]

Miller et al.

[11] **Patent Number: 5,946,647**

[45] **Date of Patent: Aug. 31, 1999**

[54] **SYSTEM AND METHOD FOR PERFORMING AN ACTION ON A STRUCTURE IN COMPUTER-GENERATED DATA**

[75] Inventors: **James R. Miller**, Mountain View; **Thomas Bonura**, Capitola; **Bonnie Nardi**, Mountain View; **David Wright**, Santa Clara, all of Calif.

[73] Assignee: **Apple Computer, Inc.**, Cupertino, Calif.

[21] Appl. No.: **08/595,257**

[22] Filed: **Feb. 1, 1996**

[51] **Int. Cl.**[6] **G06F 17/27**

[52] **U.S. Cl.** **704/9**; 704/1

[58] **Field of Search** 704/1, 7, 9–10, 704/243; 707/513, 101–104

[56] **References Cited**

U.S. PATENT DOCUMENTS

5,115,390	5/1992	Fukuda et al.	364/146
5,130,924	7/1992	Barker et al.	704/1
5,164,899	11/1992	Sobotka et al.	704/9
5,202,828	4/1993	Vertelney et al.	364/419
5,247,437	9/1993	Vale et al.	704/1
5,369,575	11/1994	Lamberti et al.	704/1
5,574,843	11/1996	Gerlach et al.	395/118

OTHER PUBLICATIONS

TerryMorse Software "What is Myrmidon" Downloaded from the Internet at URL http://www.terrymorse.com (Publication Date Unknown), 2 pages.

Shoens, K. et al. "Rufus System: Information Organization for Semi–Structured Data," Proceedings of the 19th VLDB Conference (Dublin, Ireland 1993), pp. 1–12.

Schwarz, Peter and Shoens, Kurt. "Managing Change in the Rufus System," Abstract from the IBM Almaden Research Center, pp. 1–16.

Myers, Brad A. "Tourmaline: Text Formatting by Demonstration," (Chapter 14) in *Watch What I Do: Programming by Demonstration,* edited by Allen Cypher, MIT Press, (Cambridge, MA 1993), pp. 309–321.

Maulsby, David. "Instructible Agents," Dissertation from the Department of Computer Science at The University of Calgary (Calgary, Alberta—Jun. 1994), pp. 178, 181–188, 193–196 (from Chapter 5).

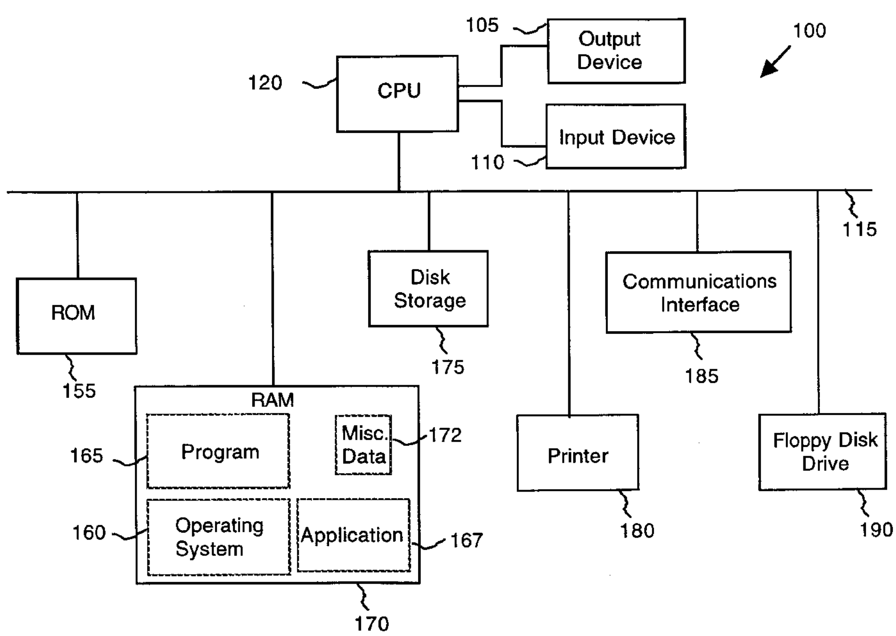
Rus, Daniela and Subramanian, Devika. "Designing Structure–Based Information Agents," AAAI Symposium (Mar. 1994), pp. 79–86.

Primary Examiner—Forester W. Isen
Assistant Examiner—Patrick N. Edouard
Attorney, Agent, or Firm—Carr & Ferrell LLP

[57] **ABSTRACT**

A system and method causes a computer to detect and perform actions on structures identified in computer data. The system provides an analyzer server, an application program interface, a user interface and an action processor. The analyzer server receives from an application running concurrently data having recognizable structures, uses a pattern analysis unit, such as a parser or fast string search function, to detect structures in the data, and links relevant actions to the detected structures. The application program interface communicates with the application running concurrently, and transmits relevant information to the user interface. Thus, the user interface can present and enable selection of the detected structures, and upon selection of a detected structure, present the linked candidate actions. Upon selection of an action, the action processor performs the action on the detected structure.

24 Claims, 10 Drawing Sheets



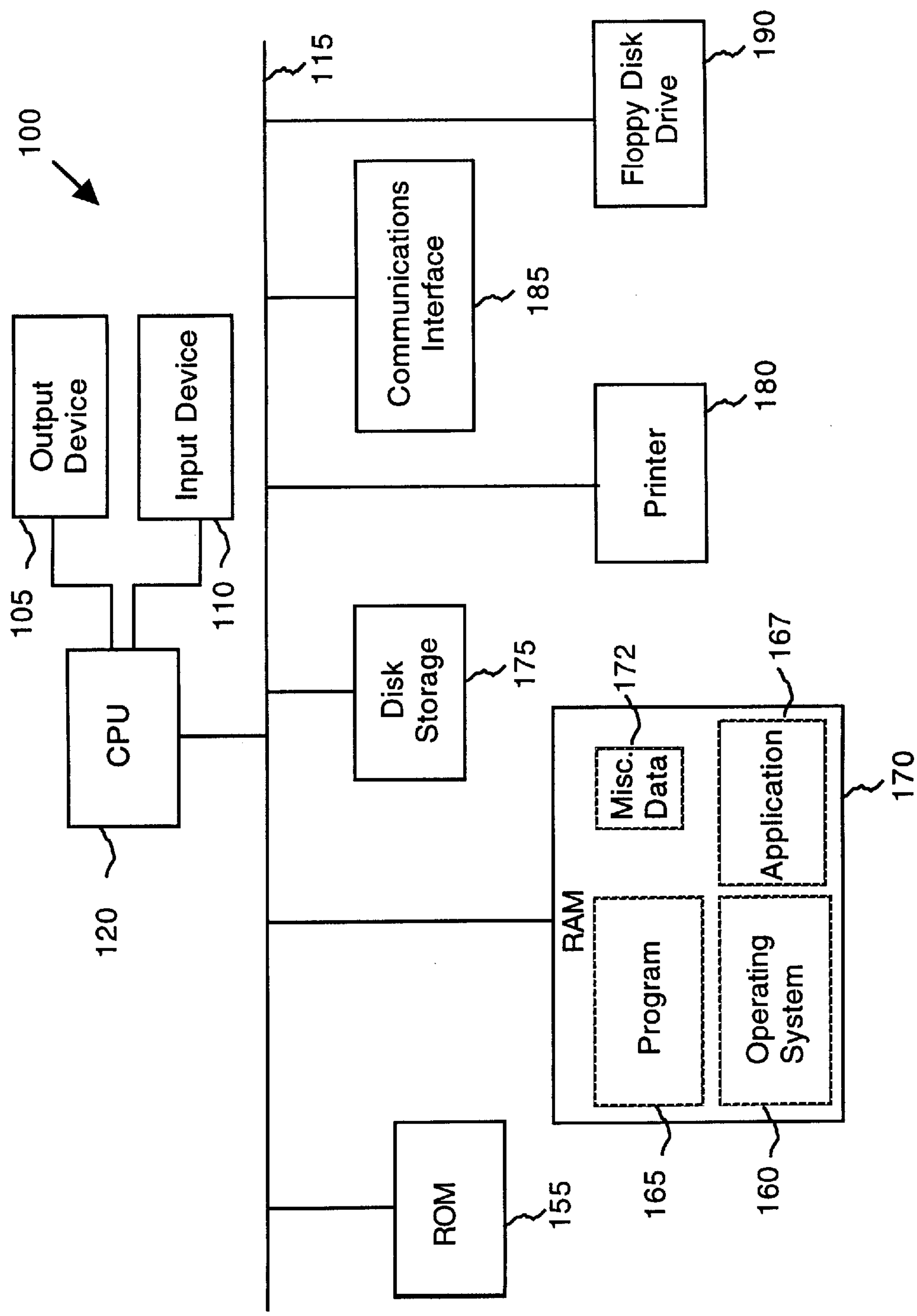

100
Output Device
Input Device
105
110
CPU
120
115
Communications Interface
185
Floppy Disk Drive
190
Printer
180
Disk Storage
175
RAM
Misc. Data
172
Application
167
170
Program
Operating System
165
160
ROM
155

FIG. 1

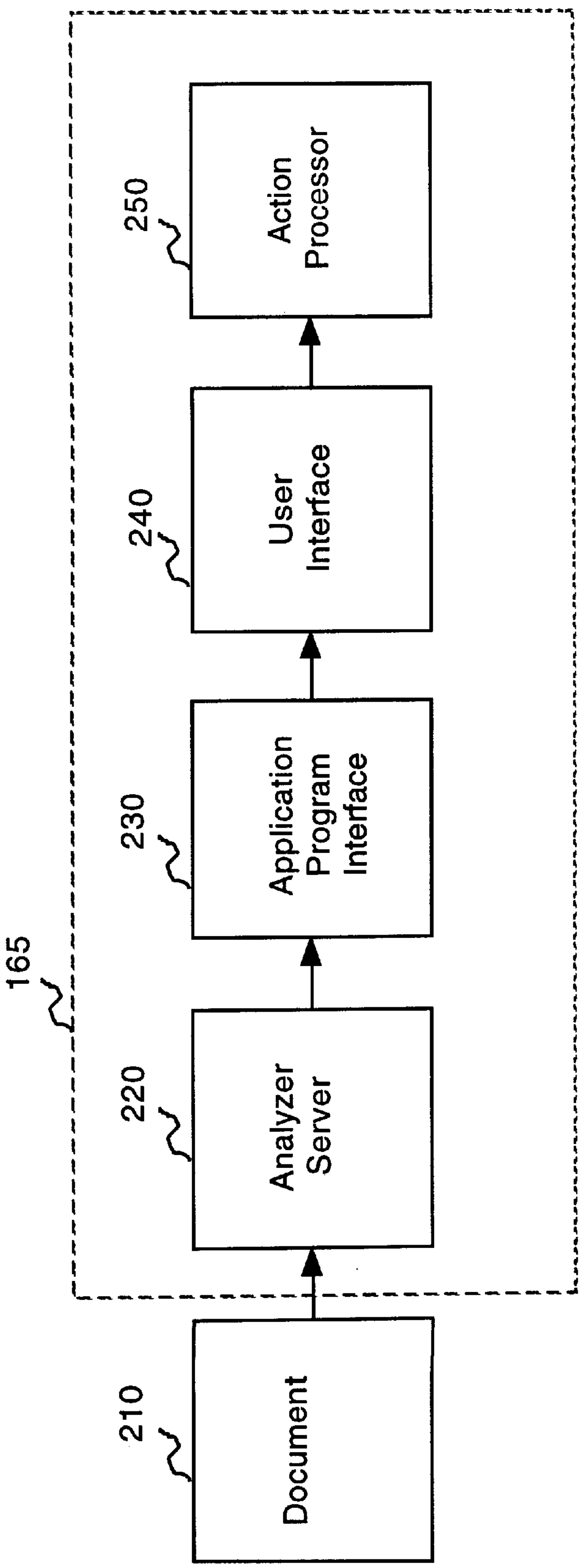

165
210
Document
220
Analyzer Server
230
Application Program Interface
240
User Interface
250
Action Processor

FIG. 2

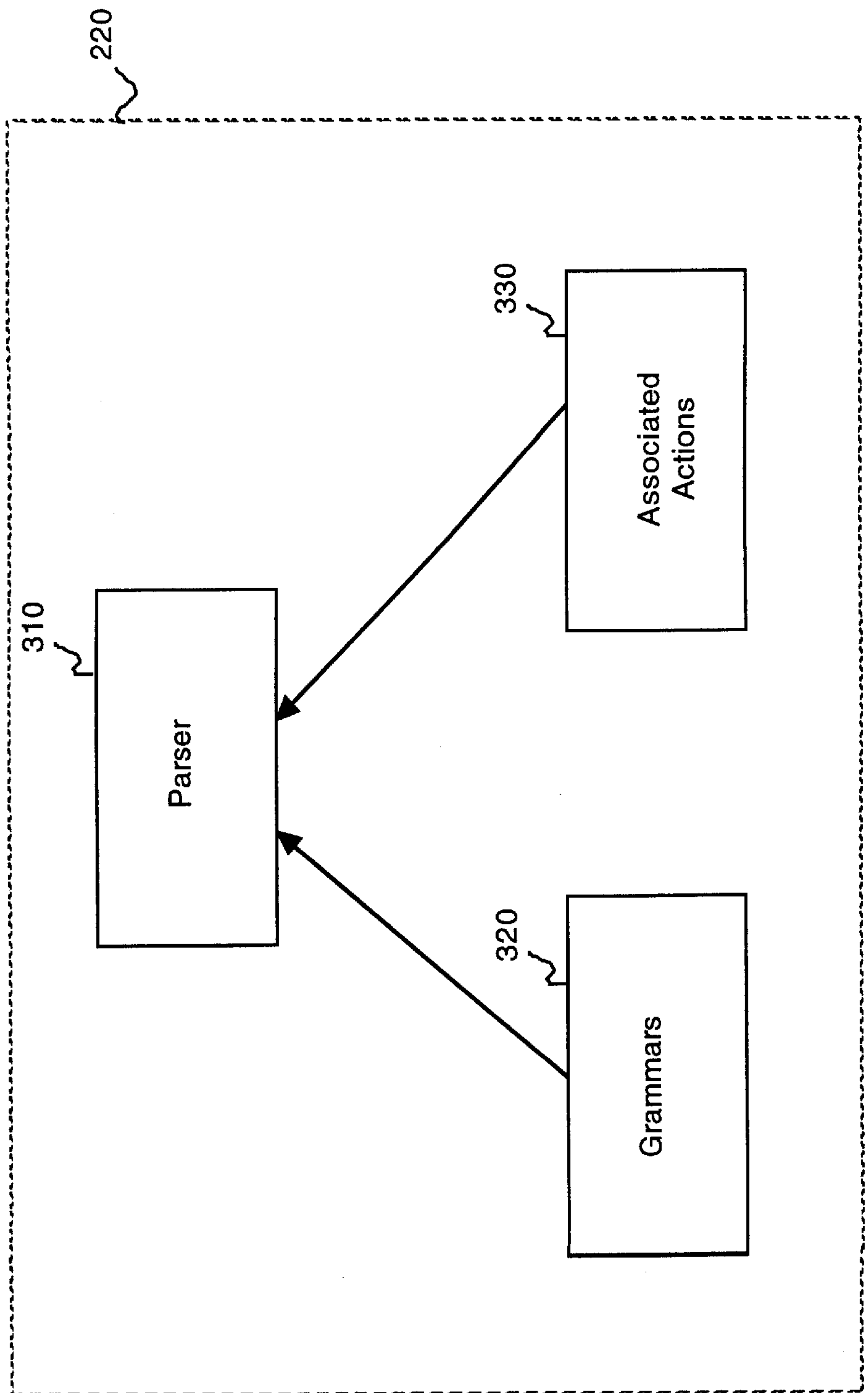


FIG. 3

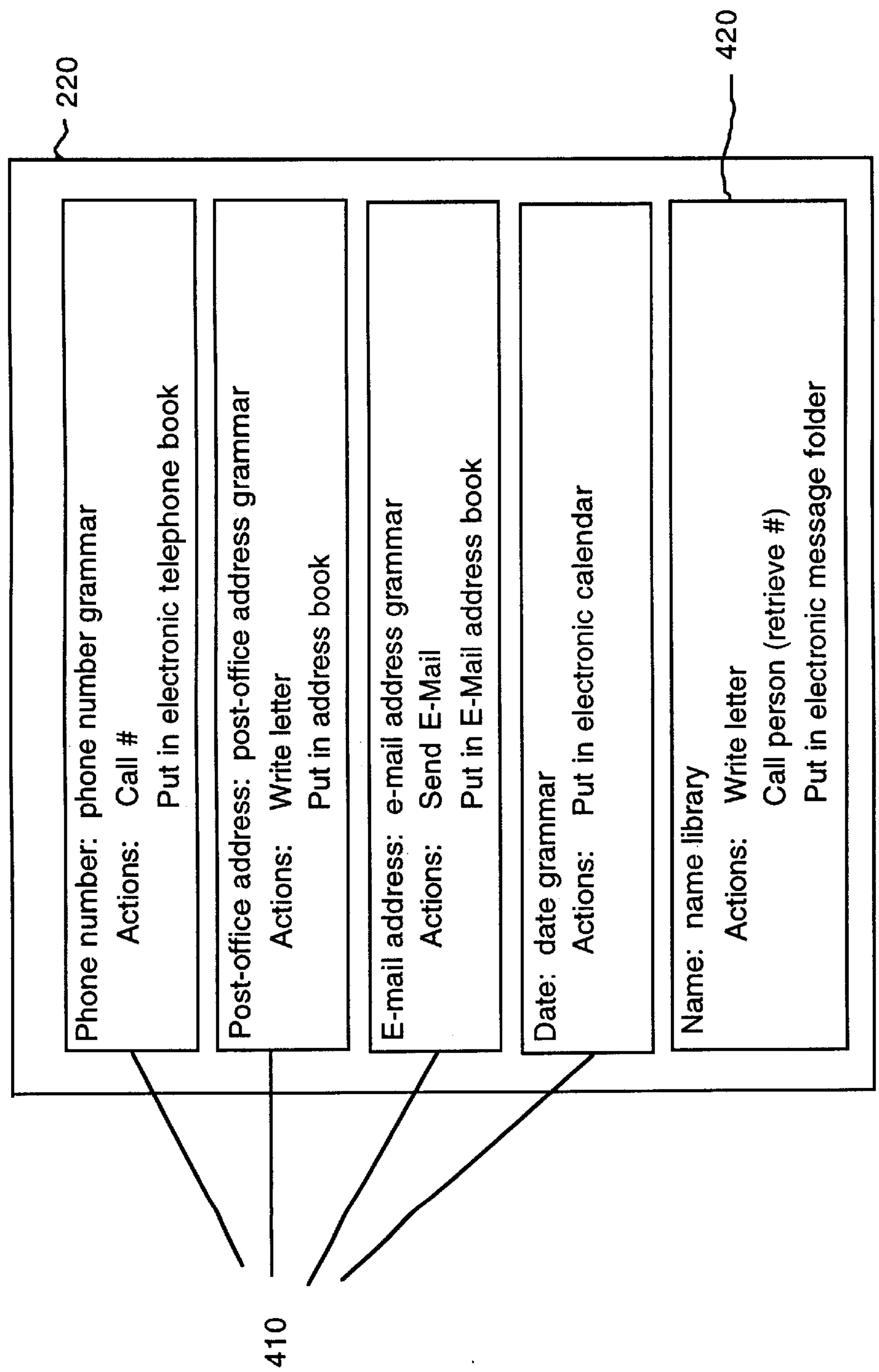

220
420
Phone number: phone number grammar
Actions: Call #
Put in electronic telephone book
Post-office address: post-office address grammar
Actions: Write letter
Put in address book
E-mail address: e-mail address grammar
Actions: Send E-Mail
Put in E-Mail address book
Date: date grammar
Actions: Put in electronic calendar
Name: name library
Actions: Write letter
Call person (retrieve #)
Put in electronic message folder
410

FIG. 4

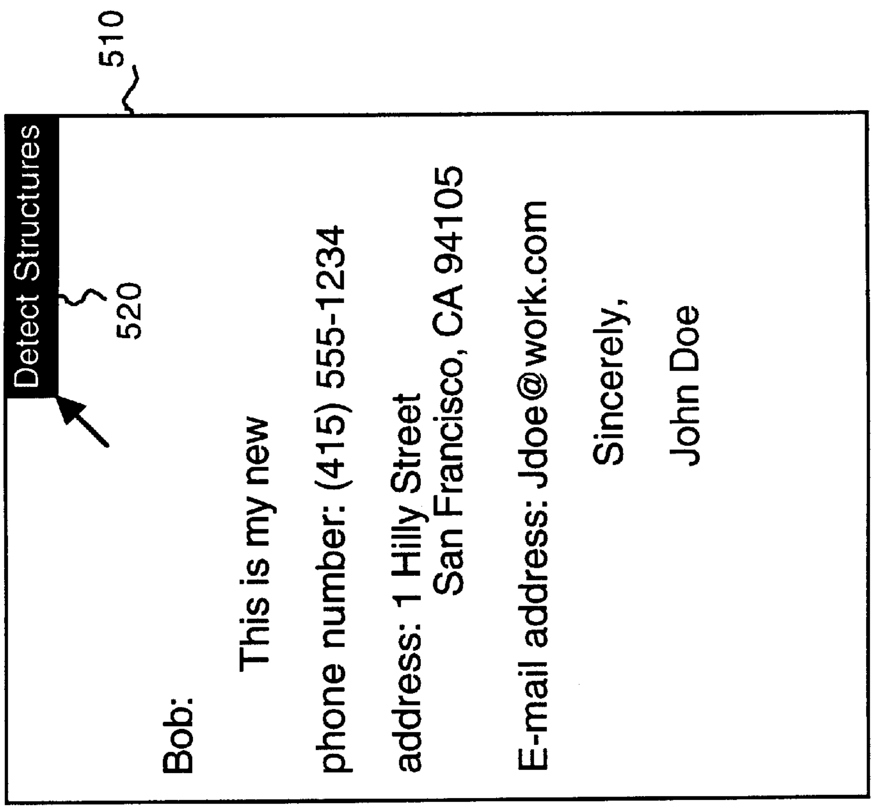

510
Detect Structures
520
Bob:
This is my new
phone number: (415) 555-1234
address: 1 Hilly Street
San Francisco, CA 94105
E-mail address: Jdoe@work.com
Sincerely,
John Doe

FIG. 5

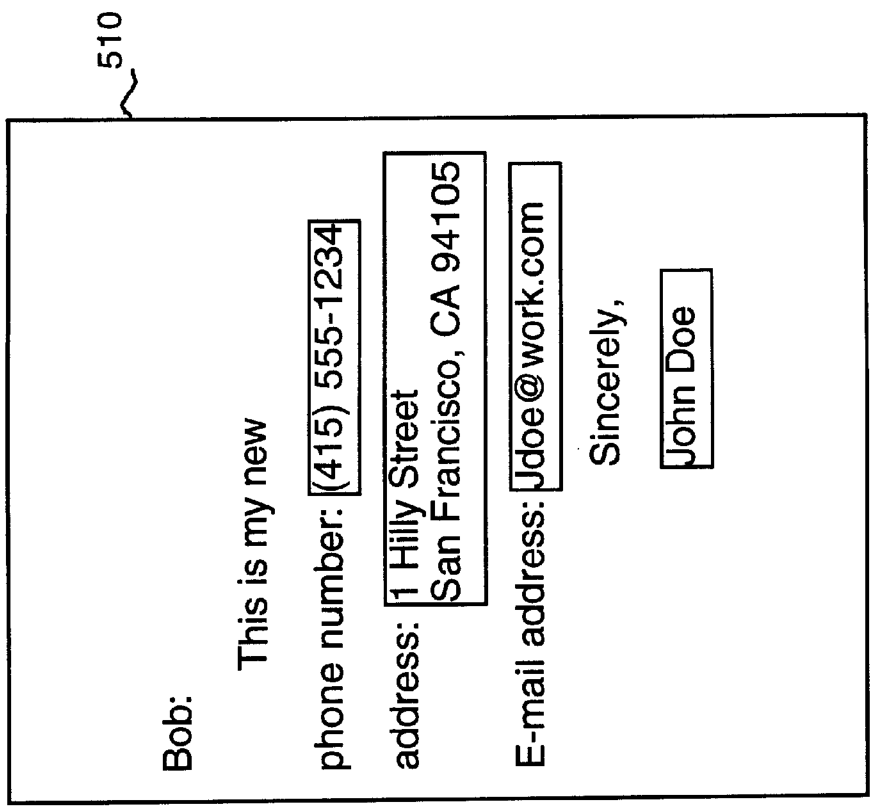

510
Bob:
This is my new
phone number: (415) 555-1234
address: 1 Hilly Street
San Francisco, CA 94105
E-mail address: Jdoe@work.com
Sincerely,
John Doe

FIG. 6

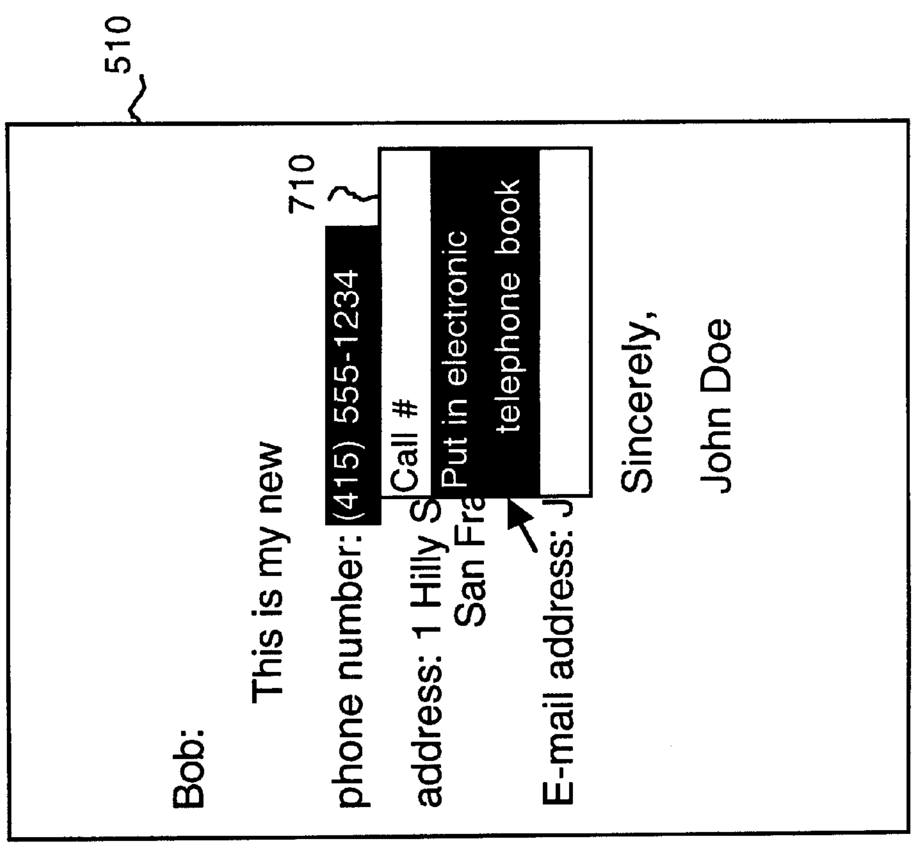

510
710
Bob:
This is my new
phone number: (415) 555-1234
Call #
Put in electronic
telephone book
address: 1 Hilly S
San Fra
E-mail address: J
Sincerely,
John Doe

FIG. 7

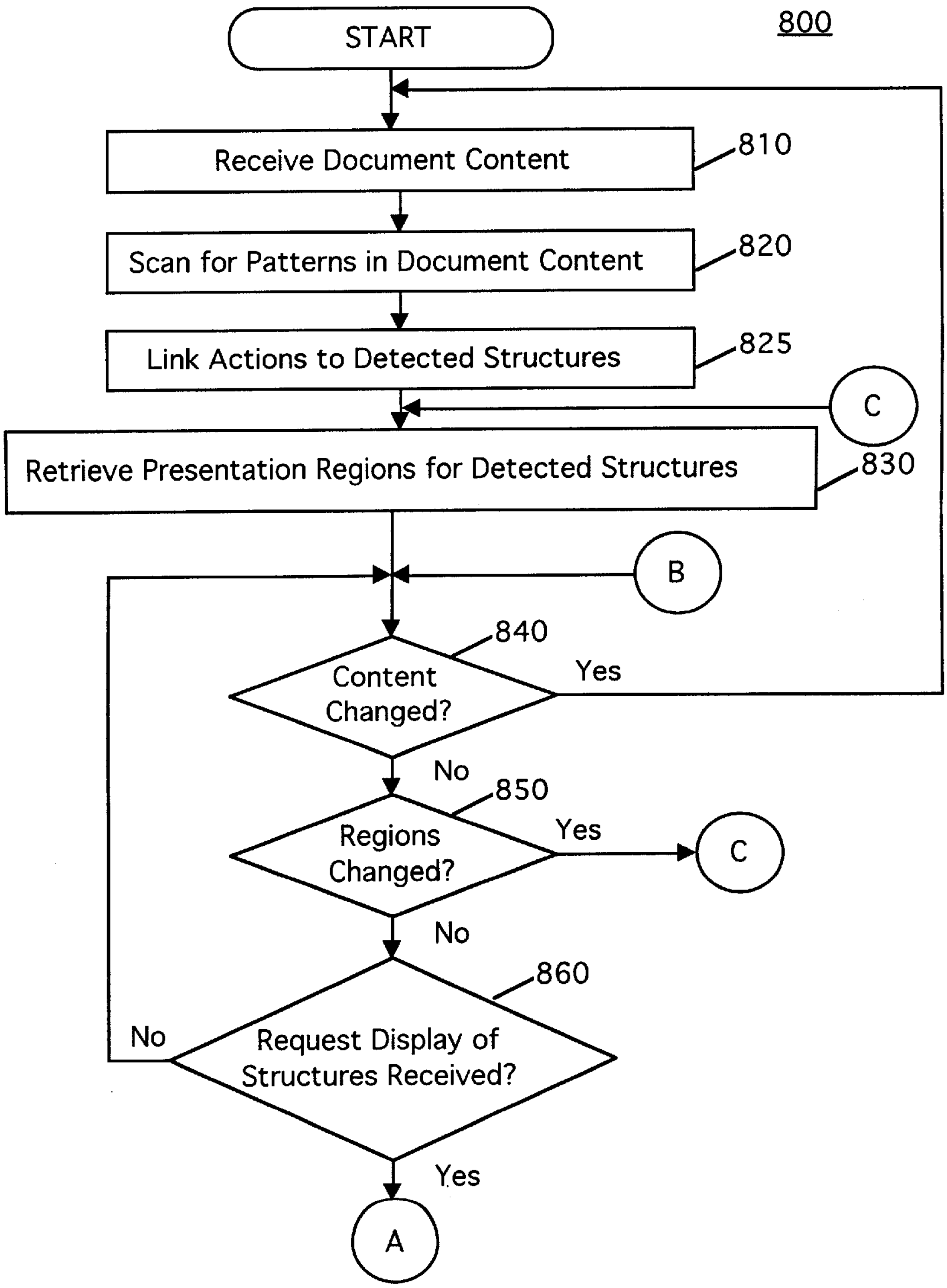

800
START
Receive Document Content
810
Scan for Patterns in Document Content
820
Link Actions to Detected Structures
825
C
Retrieve Presentation Regions for Detected Structures
830
B
840
Content Changed?
Yes
No
850
Regions Changed?
Yes
C
No
860
No
Request Display of Structures Received?
Yes
A

FIG. 8

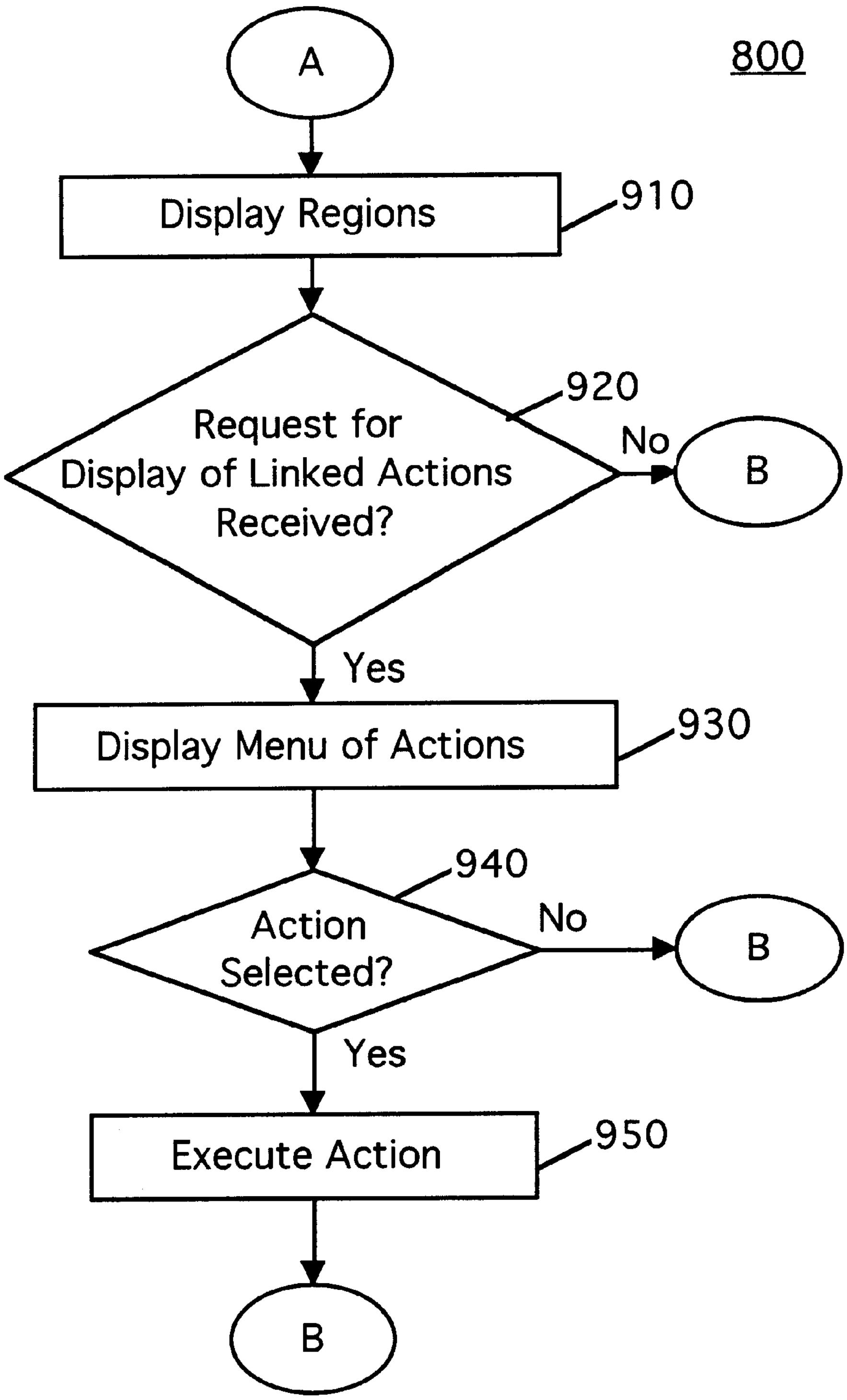

800
A
Display Regions
910
Request for Display of Linked Actions Received?
920
No
B
Yes
Display Menu of Actions
930
Action Selected?
940
No
B
Yes
Execute Action
950
B

FIG. 9

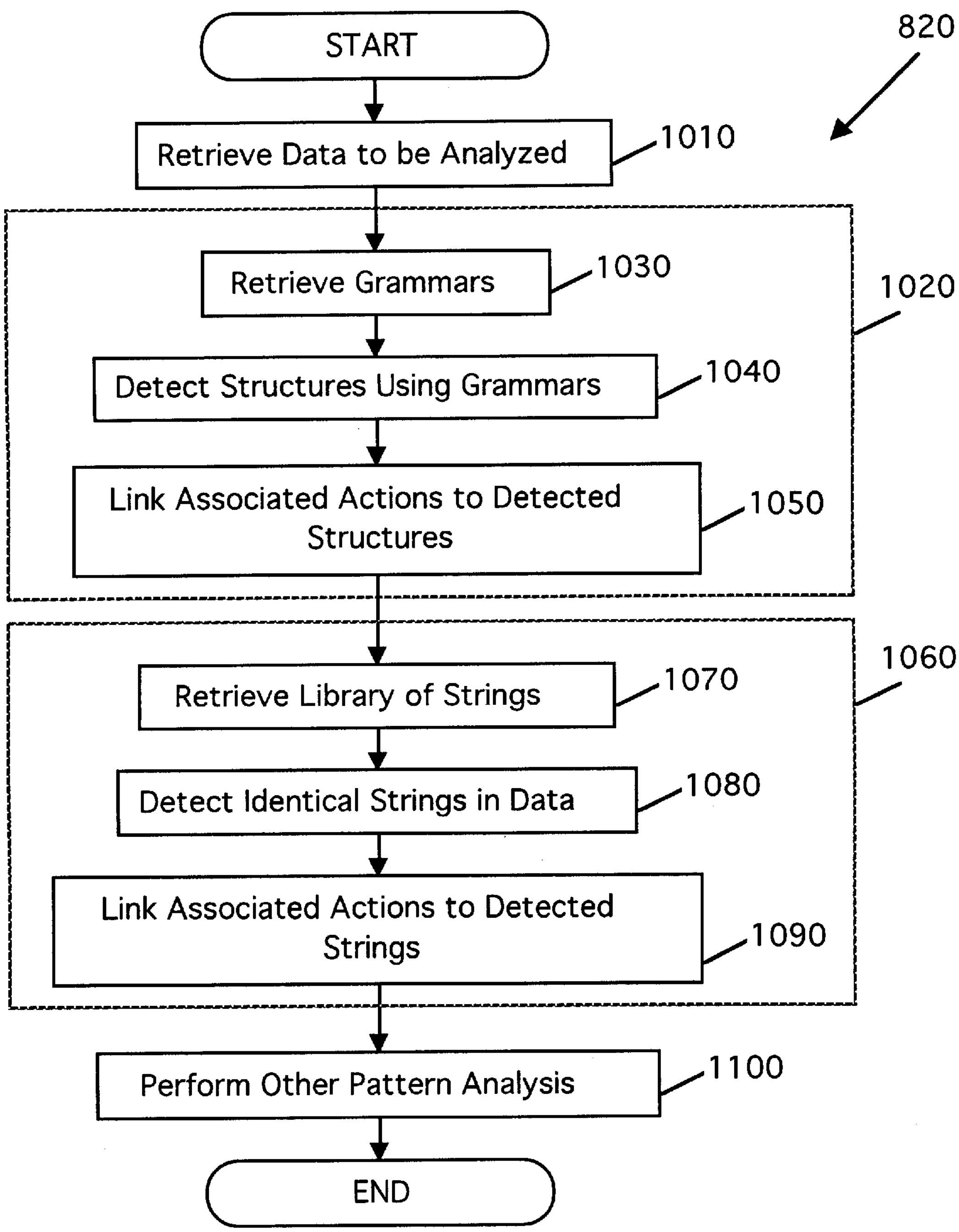

820
START
Retrieve Data to be Analyzed
1010
Retrieve Grammars
1030
1020
Detect Structures Using Grammars
1040
Link Associated Actions to Detected Structures
1050
1060
Retrieve Library of Strings
1070
Detect Identical Strings in Data
1080
Link Associated Actions to Detected Strings
1090
Perform Other Pattern Analysis
1100
END

FIG. 10

SYSTEM AND METHOD FOR PERFORMING AN ACTION ON A STRUCTURE IN COMPUTER-GENERATED DATA

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates generally to manipulation of structures in computer data. More particularly, the invention relates to a system and method for performing computer-based actions on structures identified in computer data.

2. Description of the Background Art

Much data that appears in a computer user's day-to-day activities contains recognizable structures that have semantic significance such as phone numbers, e-mail addresses, post-office addresses, zip codes and dates. In a typical day, for example, a user may receive extensive files from word-processing programs and e-mail that contain several of these structures. However, visually searching data files or documents to find these structures is laborious and cognitively disruptive, especially if the document is lengthy and hard to follow. Furthermore, missing a structure such as a date may lead to missing an important meeting or missing a deadline.

To help facilitate searching a document for these structures, programmers can create or employ pattern analysis units, such as parsers, to automatically identify the structures. For the purposes of the present description, the term "pattern" refers to data, such as a grammar, regular expression, string, etc., used by a pattern analysis unit to recognize information in a document, such as dates, addresses, phone numbers, names, etc. The term "structure" refers to an instantiation of a pattern in the document. That is, a "date" pattern will recognize the structure "Oct. 31, 1995." The application of a pattern to a document is termed "parsing."

Conventional systems that identify structures in computer data do not enable automatic performance of an action on an identified structure. For example, if a long e-mail message is sent to a user, the user may implement a pattern analysis unit to search for particular structures, such as telephone numbers. Upon identification of a structure, the user may want to perform an action on the structure, such as moving the number to an electronic telephone book. This usually involves cutting the structure from the e-mail message, locating and opening the electronic telephone book application program, pasting the structure into the appropriate field, and closing the application program. However, despite the fact that computer systems are getting faster and more efficient, this procedure is still tedious and cognitively disruptive.

One type of system that has addressed this problem involves detecting telephone numbers. Such systems enable a user to select a telephone number and request that the application automatically dial the number. However, these systems do not recognize the selected data as a telephone number, and they generally produce an error message if the user selects invalid characters as a phone number. Also, they do not enable the performance of other candidate actions, such as moving the number to an electronic telephone book. That is, if a user wishes to perform a different action on an identified telephone number, such as storing the number in an address book, the user cannot automatically perform the action but must select and transfer the number to the appropriate data base as described above.

Therefore, a system is needed that identifies structures, associates candidate actions to the structures, enables selec-

tion of an action and automatically performs the selected action on the structure.

SUMMARY OF THE INVENTION

The present invention overcomes the limitations and deficiencies of previous systems with a system that identifies structures in computer data, associates candidate actions with each detected structure, enables the selection of an action, and automatically performs the selected action on the identified structure. It will be appreciated that the system may operate on recognizable patterns for text, pictures, tables, graphs, voice, etc. So long as a pattern is recognizable, the system will operate on it. The present invention has significant advantages over previous systems, in that the present system may incorporate an open-ended number and type of recognizable patterns, an open-ended number and type of pattern analysis units, and further that the system may enable an open-ended number and type (i.e. scripts, macros, code fragments, etc.) of candidate actions to associate with, and thus perform, on each identified structure.

The present invention provides a computer system with a central processing unit (CPU), input/output (I/O) means, and a memory that includes a program to identify structures in a document and perform selected computer-based actions on the identified structures. The program includes program subroutines that include an analyzer server, an application program interface, a user interface and an action processor. The analyzer server receives data from a document having recognizable structures, and uses patterns to detect the structures. Upon detection of a structure, the analyzer server links actions to the detected structure. Each action is a computer subroutine that causes the CPU to perform a sequence of operations on the particular structure to which it is linked. An action may specify opening another application, loading the identified structure into an appropriate field, and closing the application. An action may further include internal actions, such as storing phone numbers in an electronic phone book, addresses in an electronic address book, appointments on an electronic calendar, and external actions such as returning phone calls, drafting letters, sending facsimile copies and e-mail, and the like.

Since the program may be executed during the run-time of another program, i.e. the application which presents the document, such as Microsoft Word, an application program interface provides mechanisms for interprogram communications. The application program interface retrieves and transmits relevant information from the other program to the user interface for identifying, presenting and enabling selection of detected structures. Upon selection of a detected structure, the user interface presents and enables selection of candidate actions. When a candidate action is selected, the action processor performs the selected action on the selected structure.

In addition to the computer system, the present invention also provides methods for performing actions on identified structures in a document. In this method, the document is analyzed using a pattern to identify corresponding structures. Identified structures are stored in memory and presented to the user for selection. Upon selection of an identified structure, a menu of candidate actions is presented, each of which may be selected and performed on the selected structure.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is a block diagram of a computer system having a program stored in RAM, in accordance with the present invention.

FIG. **2** is a block diagram of the program of FIG. **1**.

FIG. **3** is a block diagram illustrating the analyzer server of FIG. **2**.

FIG. **4** is a block diagram illustrating a particular example of the analyzer server of FIG. **2**.

FIG. **5** illustrates a window presenting an example of a document having recognizable structures.

FIG. **6** illustrates a window with the identified structures in the example document of FIG. **5** highlighted based on the analyzer server of FIG. **4**.

FIG. **7** illustrates a window showing the display of a pop-up menu for selecting an action.

FIGS. **8** and **9** together are a flowchart depicting the preferred method for selecting and performing an action on an identified structure.

FIG. **10** is a flowchart depicting the preferred method for identifying a structure in a data sample.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

Referring now to FIG. **1**, a block diagram is shown of a computer system **100** including a CPU **120**. Computer system **100** is preferably a microprocessor-based computer, such as a Power Macintosh manufactured by Apple Computer, Inc. of Cupertino, Calif. An input device **110**, such as a keyboard and mouse, and an output device **105**, such as a CRT or voice module, are coupled to CPU **120**. ROM **155**, RAM **170** and disk storage **175** are coupled to CPU **120** via signal bus **115**. Computer system **100** optionally further comprises a printer **180**, a communications interface **185**, and a floppy disk drive **190**, each coupled to CPU **120** via signal bus **115**.

Operating system **160** is a program that controls and facilitates the processing carried out by CPU **120**, and is typically stored in RAM **170**. Application **167** is a program, such as a word-processor or e-mail program, that presents data on output device **105** to a user. The program **165** of the present invention is stored in RAM **170** and causes CPU **120** to identify structures in the data presented by application **167**, to associate actions with the structures identified in the data, to enable the user to select a structure and an action, and to automatically perform the selected action on the identified structure. This program **165** may be stored in disk storage **175** and loaded into an allocated section of RAM **170** prior to execution by CPU **120**. Another section of RAM **170** is used for storing intermediate results and miscellaneous data **172**. Floppy disk drive **190** enables the storage of the present program **165** onto a removable storage medium which may be used to initially load program **165** into computer system **100**.

Referring now to FIG. **2**, a schematic block diagram of program **165** is shown together with its interaction with a document **210**. Program **165** contains program subroutines including an analyzer server **220**, an application program interface **230**, a user interface **240** and an action processor **250**. Analyzer server **220** receives data having recognizable patterns from a document **210**, which may be retrieved from a storage medium such as RAM **170**, ROM **155**, disk storage **175**, or the like, and presented on output device **105** by application **167**. Analyzer server **220** comprises one or more pattern analysis units, such as a parser and grammars or a fast string search function and dictionaries, which uses patterns to parse document **210** for recognizable structures. Upon detection of a structure, analyzer server **220** links actions associated with the responsible pattern to the detected structure, using conventional pointers.

After identifying structures and linking actions, application program interface **230** communicates with application **167** to obtain information on the identified structures so that user interface **240** can successfully present and enable selection of the actions. In a display-type environment, application program interface **230** retrieves the locations in document **210** of the presentation regions for the detected structures from application **167**. Application program interface **230** then transmits this location information to user interface **240**, which highlights the detected structures, although other presentation mechanisms can be used. User interface **240** enables selection of an identified structure by making the presentation regions mouse-sensitive, i.e. aware when a mouse event such as a mouse-down operation is performed while the cursor is over the region. Alternative selection mechanisms can be used such as touch sensitive screens and dialog boxes. It will be appreciated that detected structures can be hierarchical, i.e. that a sub-structure can itself be selected and have actions associated with it. For example, a user may be able to select the year portion of an identified date, and select actions specific to the year rather than to the entire date.

User interface **240** communicates with application **167** through application program interface **230** to determine if a user has performed a mouse-down operation in a particular mouse-sensitive presentation region, thereby selecting the structure presented at those coordinates. Upon selection of this structure, user interface **240** presents and enables selection of the linked candidate actions using any selection mechanism, such as a conventional pull-down or pop-up menu.

The above description of the user interface is cast in terms of a purely visual environment. However, the invention is not limited to visual interface means. For example, in an audio environment, user interface **240** may present the structures and associated actions to the user using voice synthesis and may enable selection of a pattern and action using voice or sound activation. In this type of embodiment, analyzer server **220** may be used in conjunction with a text-to-speech synthesis application **167** that reads documents to users over a telephone. Analyzer server **220** scans document **210** to recognize patterns and link actions to the recognized patterns in the same manner as described above. In the audio environment, user interface **240** may provide a special sound after application **167** reads a recognized pattern, and enable selection of the pattern through the use of an audio interface action, such as a voice command or the pressing of a button on the touch-tone telephone keypad as before. Thus, user interface **240** may present the linked actions via voice synthesis. One can create various environments having a combination of sensory mechanisms.

Upon selection of a candidate action, user interface **240** transmits the selected structure and the selected action to action processor **250**. Action processor **250** retrieves the sequence of operations that constitute the selected action, and performs the sequence using the selected structure as the object of the selected action.

Referring now to FIG. **3**, a block diagram illustrating an analyzer server **220** is shown. In this figure, analyzer server **220** is described as having a parser **310** and a grammar file **320**, although alternatively or additionally a fast string search function or other function can be used. Parser **310** retrieves a grammar from grammar file **320** and parses text using the retrieved grammar. Upon identification of a structure in the text, parser **310** links the actions associated with the grammar to the identified structure. More particularly, parser **310** retrieves from grammar file **320** pointers attached

to the grammar and attaches the same pointers to the identified structure. These pointers direct the system to the associated actions contained in associated actions file **330**. Thus, upon selection of the identified structure, user interface **240** can locate the linked actions.

FIG. **4** illustrates an example of an analyzer server **220**, which includes grammars **410** and a string library **420** such as a dictionary, each with associated actions. One of the grammars **410** is a telephone number grammar with associated actions for dialing a number identified by the telephone number grammar or placing the number in an electronic telephone book. Analyzer server **220** also includes grammars for post-office addresses, e-mail addresses and dates, and a string library **420** containing important names. When analyzer server **220** identifies an address using the "e-mail address" grammar, actions for sending e-mail to the identified address and putting the identified address in an e-mail address book are linked to the address.

FIG. **5** shows a window **510** presenting an exemplary document **210** having data containing recognizable structures, including a phone number, post-office address, e-mail address, and name. Window **510** includes a button **520** for initiating program **165**, although alternative mechanisms such as depressing the "option" key may be used. Upon initiation of program **165**, system **100** transmits the contents of document **210** to analyzer server **220**, which parses the contents based on grammars **410** and strings **420** (FIG. **4**). This parsing process produces the window shown in FIG. **6**. As illustrated in FIG. **6**, analyzer server **220** identifies the phone number, post-office address, e-mail address and name. Although not shown in FIG. **6**, analyzer server **220** links the actions associated with grammars **410** and strings **420** to these identified structures, and application program interface **230** retrieves information on the location of these structures from application **167**. User interface **240** then highlights the identified structures in document **210**, and makes the identified structures mouse-sensitive.

As shown in FIG. **7**, upon recognition of a mouse-down operation over a structure, user interface **240** presents a pop-up menu **710**. In this example, pop-up menu **710** displays the candidate actions linked to the selected telephone number grammar **410**, including dialing the number and putting the number into an electronic telephone book. Upon selection of the action for putting the number in an electronic telephone book, user interface **240** transmits the corresponding telephone number and selected action to action processor **250**. Action processor **250** locates and opens the electronic telephone book, places the telephone number in the appropriate field and allows the user to input any additional information into the file.

FIGS. **8** and **9** display a flowchart illustrating preferred method **800** for recognizing patterns in documents and performing actions. This method is carried out during the run-time of application **167**. Referring first to FIG. **8**, method **800** starts by receiving **810** the content, or a portion of the content, from document **210**. Assuming program **165** initiates with the receipt of any text, the received content or portion is scanned **820** for identifiable structures using the patterns in analyzer server **220**. Upon detection of a structure based on a particular pattern, actions associated with the particular pattern are linked **825** to the detected structure. Assuming a display-type environment, the presentation region location for a detected structure is retrieved **830** from application **167**. If the document content being displayed on output device **105** is changed **840**, for example by the user adding or modifying text, method **800** restarts. Otherwise, method **800** continues with block **850**. If the presentation

regions change **850**, for example by the a user scrolling document **210**, then new presentation regions from application **167** are again retrieved **830**. Otherwise, method **800** continues to block **860**. As illustrated by block **860**, method **800** loops between blocks **840** and **860** until a request for display of identified structures is received **860**. It will be appreciated that the steps of the loop (blocks **840**, **850** and **860**) can be performed by application **167**.

Referring also to FIG. **9**, when a request for the display of detected structures is received **860**, the regions are displayed **910** using presentation mechanisms such as highlighting the presentation region around each detected structure, although alternative presentation mechanisms can be used. If a request for the display of candidate actions linked to a detected structure is not received **920**, method **800** returns to block **840**. However, if a request is received **920**, the actions linked in block **825** are displayed **930**. This request for display of candidate actions can be performed using a selection mechanism, such as a mouse-down operation over a detected structure, which causes the candidate actions linked to the structure to be displayed **930**. Display **930** of candidate actions may be implemented using a pop-up menu, although alternative presentation mechanisms can be used such as pull-down menus, dialog boxes and voice synthesizers.

As illustrated in block **940**, if an action from the displayed candidate actions is not selected **940**, method **800** returns to block **840**. However, if an action is selected **940**, the action is executed **950** on the structure selected in block **920**. After execution **950** of an action, method **800** returns to block **840**. Method **800** ends when the user exits application **167**, although other steps for ending method **800** can alternatively be used.

Referring now to FIG. **10**, a flowchart illustrating the preferred method **820** for scanning and detecting patterns in a document is shown. Method **820** starts by retrieving **1010** data to be analyzed. After the data is retrieved, several pattern analysis processes may be performed on the data. As illustrated in block **1020**, a parsing process retrieves **1030** grammars, detects **1040** structures in the data based on the retrieved grammars, and links **1050** actions associated with each grammar to each structure detected by that grammar. As illustrated in block **1060**, a fast string search function retrieves **1070** the contents of string library **420**, detects **1080** the strings in the data identical to those in the string library **420**, and links **1090** actions associated with the library string to the detected string. As illustrated in block **1100**, additional pattern analysis processes, such as a neural net scan, can be performed **1100** to detect in the data other patterns, such as pictures, graphs, sound, etc. Method **820** then ends. Alternatively, the pattern analysis processes can be performed in parallel using a multiprocessor multitasking system, or using a uniprocessor multithreaded multitasking system where a thread is allocated to execute each pattern detection scheme.

These and other variations of the preferred and alternate embodiments and methods are provided by the present invention. For example, program **165** in FIG. **1** can be stored in ROM, disk, or in dedicated hardware. In fact, it may be realized as a separate electronic circuit. Other components of this invention may be implemented using a programmed general purpose digital computer, using application specific integrated circuits, or using a network of interconnected conventional components and circuits. The analyzer server **220** of FIG. **2** may use a neural net for searching a graphical document **210** for faces, or a musical library for searching a stored musical piece **210** for sounds. The user interface **240**

may present structures and actions via voice synthesis over a telephone line connection to system **100**. The embodiments described have been presented for purposes of illustration and are not intended to be exhaustive or limiting, and many variations and modifications are possible in light of the foregoing teaching. The system is limited only by the following claims.

What is claimed is:

1. A computer-based system for detecting structures in data and performing actions on detected structures, comprising:

an input device for receiving data;

an output device for presenting the data;

a memory storing information including program routines including

an analyzer server for detecting structures in the data, and for linking actions to the detected structures;

a user interface enabling the selection of a detected structure and a linked action; and

an action processor for performing the selected action linked to the selected structure; and

a processing unit coupled to the input device, the output device, and the memory for controlling the execution of the program routines.

2. The system recited in claim **1**, wherein the analyzer server stores detected structures in the memory.

3. The system recited in claim **1**, wherein the input device receives the data from an application running concurrently, and wherein the program routines stored in memory further comprise an application program interface for communicating with the application.

4. The system recited in claim **1**, wherein the analyzer server includes grammars and a parser for detecting structures in the data.

5. The system recited in claim **4**, wherein the analyzer server includes actions associated with each of the grammars, and wherein the analyzer server links to a detected structure the actions associated with the grammar which detects that structure.

6. The system recited in claim **1**, wherein the analyzer server includes a string library and a fast string search function for detecting string structures in the data.

7. The system recited in claim **6**, wherein the analyzer server includes actions associated with each of the strings, and wherein the analyzer server links to a detected structure the actions associated with the grammar which detects that string structure.

8. The system recited in claim **1**, wherein the user interface highlights detected structures.

9. The system recited in claim **1**, wherein the user interface enables selection of an action by causing the output device to display a pop-up menu of the linked actions.

10. The system recited in claim **1**, wherein the programs stored in the memory further comprise an application running concurrently that causes the output device to present the data received by the input device, and an application program interface that provides interrupts and communicates with the application.

11. The system recited in claim **1**, wherein the user interface enables the selection of a detected structure and a linked action using sound activation.

12. The system recited in claim **1**, wherein a first one of the actions may invoke a second one of the actions.

13. A program storage medium storing a computer program for causing a computer to perform the steps of:

receiving computer data;

detecting a structure in the data;

linking at least one action to the detected structure;

enabling selection of the structure and a linked action; and

executing the selected action linked to the selected structure.

14. In a computer having a memory storing actions, a system for causing the computer to perform an action on a structure identified in computer data, comprising:

means for receiving computer data;

means for detecting a structure in the data;

means for linking at least one action to the detected structure;

means for selecting the structure and a linked action; and

means for executing the selected action linked to the selected structure.

15. In a computer having a memory storing actions, a method for causing the computer to perform an action on a structure identified in computer data, comprising the steps of:

receiving computer data;

detecting a structure in the data;

linking at least one action to the detected structure;

enabling selection of the structure and a linked action; and

executing the selected action linked to the selected structure.

16. The method recited in claim **15**, wherein the computer data is received from the application running concurrently.

17. The method recited in claim **15**, wherein the memory contains grammars, and wherein the step of detecting a structure further comprises the steps of retrieving a grammar and parsing the data based on the grammar.

18. The method recited in claim **17**, wherein the grammar is associated with a particular action, and wherein the step of linking at least one action to the detected structure includes the step of linking the particular action to the detected structure.

19. The method recited in claim **15**, wherein the memory contains strings, and wherein the step of detecting a structure further comprises the steps of retrieving a string from the memory and scanning the data to identify the string.

20. The method recited in claim **15**, further comprising after the step of detecting a structure, the step of highlighting the detected structure.

21. The method recited in claim **15**, further comprising, after the step of linking at least one action to the detected structure, the step of displaying and enabling selection of an action for performance on the detected structure.

22. A computer-based method for causing a computer to identify, select and perform an action on a structure in computer data received from a concurrently running application, said application presenting the computer data to the user, the method comprising the steps of:

receiving computer data from the application;

detecting a structure in the computer data;

linking at least one action to the detected structure;

communicating with the application to determine the location of the detected structure as presented by the application, to enable selection of the detected structure and a linked action, and to determine if the detected structure and a linked action have been selected; and

performing a selected action linked to the detected pattern.

23. The method recited in claim **15**, wherein the step of enabling uses sound activation.

24. The method recited in claim **15**, wherein a first one of the actions may invoke a second one of the actions.

* * * * *

EXHIBIT I

THE UNITED STATES OF AMERICA

TO ALL TO WHOM THESE PRESENTS SHALL COME:

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office

November 10, 2009

THIS IS TO CERTIFY THAT ANNEXED HERETO IS A TRUE COPY FROM THE RECORDS OF THIS OFFICE OF:

U.S. PATENT: *5,929,852*
ISSUE DATE: *July 27, 1999*

By Authority of the
Under Secretary of Commerce for Intellectual Property
and Director of the United States Patent and Trademark Office

T. Lawrence

T. LAWRENCE
Certifying Officer

US005929852A

United States Patent [19]

Fisher et al.

[11] **Patent Number: 5,929,852**

[45] **Date of Patent: Jul. 27, 1999**

[54] **ENCAPSULATED NETWORK ENTITY REFERENCE OF A NETWORK COMPONENT SYSTEM**

[75] Inventors: **Stephen Fisher; Michael A. Cleron**, both of Menlo Park; **Timo Bruck**, Mountian View, all of Calif.

[73] Assignee: **Apple Computer, Inc.**, Cupertino, Calif.

[21] Appl. No.: **09/007,691**

[22] Filed: **Jan. 15, 1998**

Related U.S. Application Data

[63] Continuation of application No. 08/435,880, May 5, 1995, abandoned.

[51] **Int. Cl.**[6] **G06T 1/00**

[52] **U.S. Cl.** **345/335**

[58] **Field of Search** 345/335, 339, 345/348, 356; 395/701, 200.47, 200.48, 680, 681, 682, 683, 684

[56] **References Cited**

U.S. PATENT DOCUMENTS

5,202,828	4/1993	Vertelney et al.	395/936 X
5,481,666	1/1996	Nguyen et al.	395/762
5,500,929	3/1996	Dickinson	395/160
5,530,852	6/1996	Meske, Jr. et al.	395/600
5,537,546	7/1996	Sauter	395/762
5,548,722	8/1996	Jalalian et al.	395/200.1
5,574,862	11/1996	Marianetti, II	395/200.08
5,659,791	8/1997	Nakajima et al.	345/302
5,724,506	3/1998	Cleron et al.	395/200.01
5,724,556	3/1998	Souder et al.	345/335 X
5,781,189	7/1998	Holleran et al.	345/335
5,819,090	10/1998	Wolf et al.	345/335 X

FOREIGN PATENT DOCUMENTS

WO A9107719 5/1991 WIPO .

OTHER PUBLICATIONS

Develop, The Apple Technical Journal, "Building an OpenDoc Part Handler", Issue 19, Sep., 1994, pp. 6–16.

Baker, S. "Mosaic—Surfing at Home and Abroad," Proceedings ACM SIGUCCS User Services Conference XXII, Oct. 16–19, 1994, pp. 159–163.

PCT International Search Report dated Oct. 22, 1996 in corresponding PCT Case No. PCT/US96/06376.

MacWeek, Nov. 7, 1994, vol. 8, No. 44, Cyberdog to Fetch Internet Resources for OpenDoc APPS, Robert Hess.

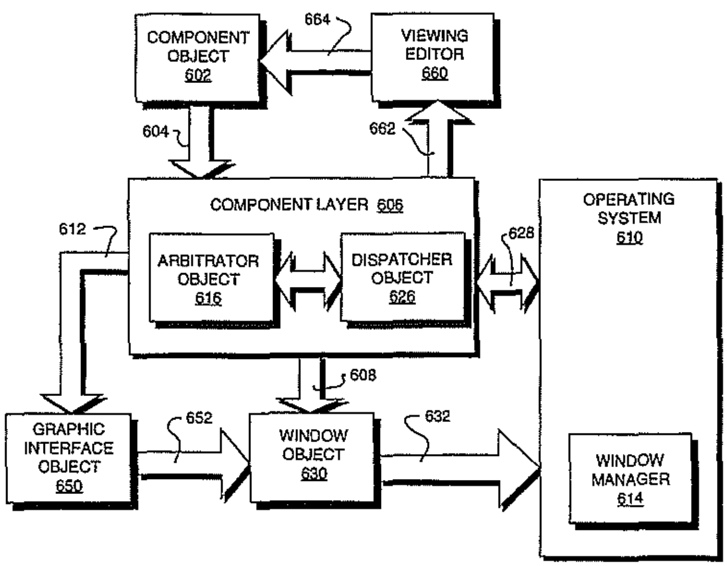
Opinion, MacWeek Nov. 14, 1994, The Second Decade, Cyberdog Could Be a Breakthrough if it's Kept on a Leash, Henry Norr.

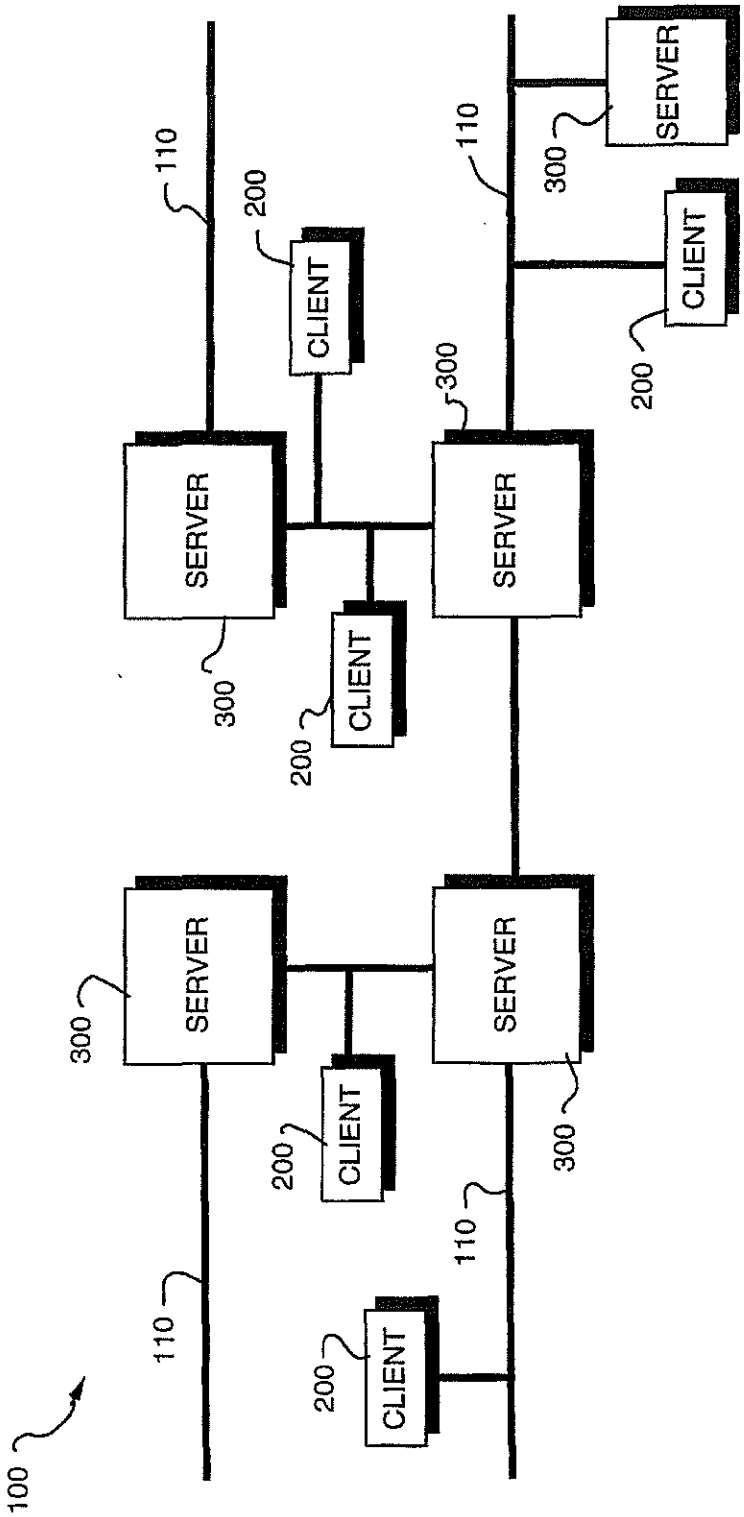
Primary Examiner—Joseph H. Feild
Attorney, Agent, or Firm—Cesari & McKenna, LLP

[57] **ABSTRACT**

A network-oriented component system efficiently accesses information from a network resource located on a computer network by creating an encapsulated network entity that contains a reference to that resource. The encapsulated entity is preferably implemented as a network component stored on a computer remotely displaced from the referenced resource. In addition, the encapsulated entity may be manifested as a visual object on a graphical user interface of a computer screen. Such visual manifestation allows a user to easily manipulate the entity in order to display the contents of the resource on the screen or to electronically forward the entity over the network.

20 Claims, 14 Drawing Sheets





100
110
110
110
110
200
200
200
200
200
300
300
300
300
300
SERVER
SERVER
SERVER
SERVER
SERVER
CLIENT
CLIENT
CLIENT
CLIENT
CLIENT

FIG. 1

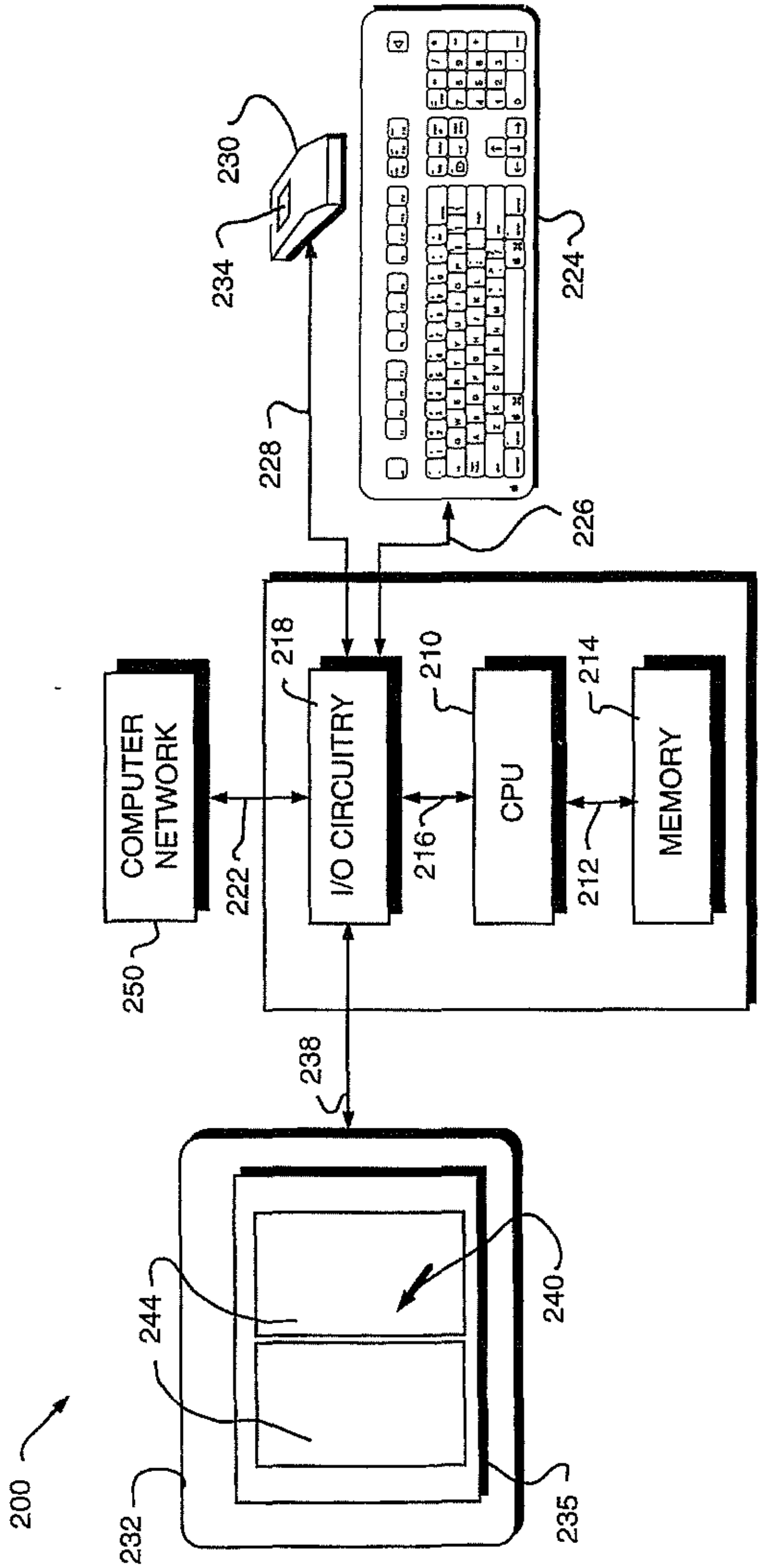

COMPUTER NETWORK
I/O CIRCUITRY
CPU
MEMORY
200
210
212
214
216
218
222
224
226
228
230
232
234
235
238
240
244
250

FIG. 2

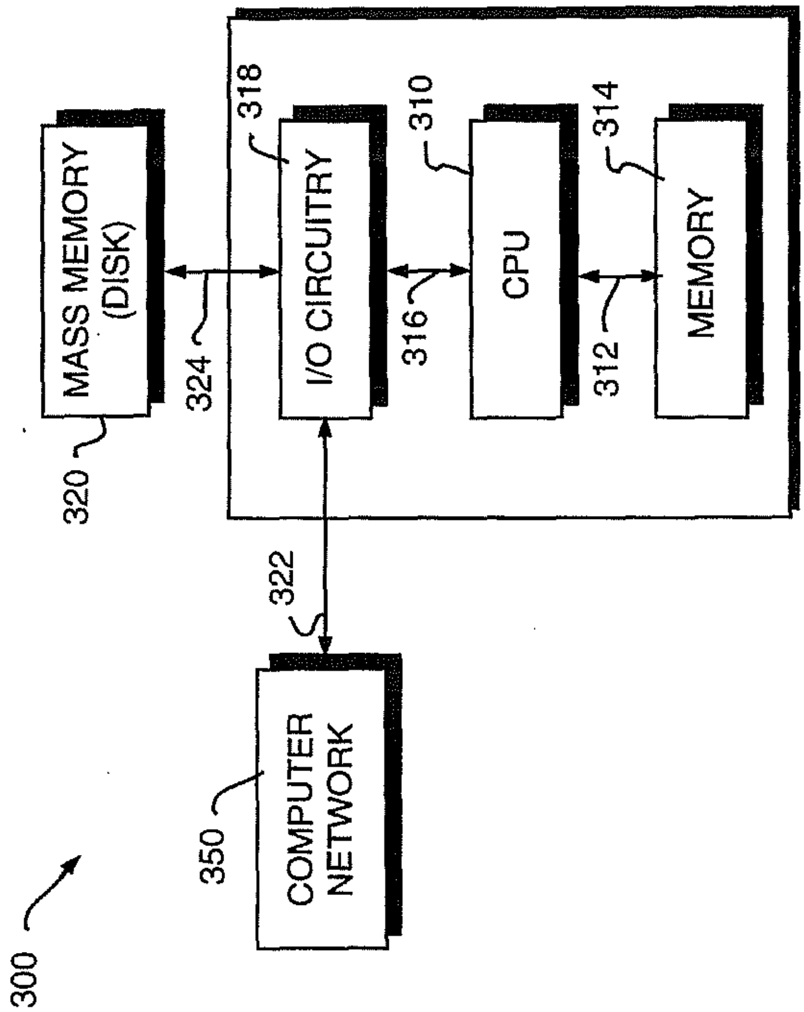

MASS MEMORY (DISK)
320
324
I/O CIRCUITRY
318
316
CPU
310
312
MEMORY
314
322
COMPUTER NETWORK
350
300

FIG. 3

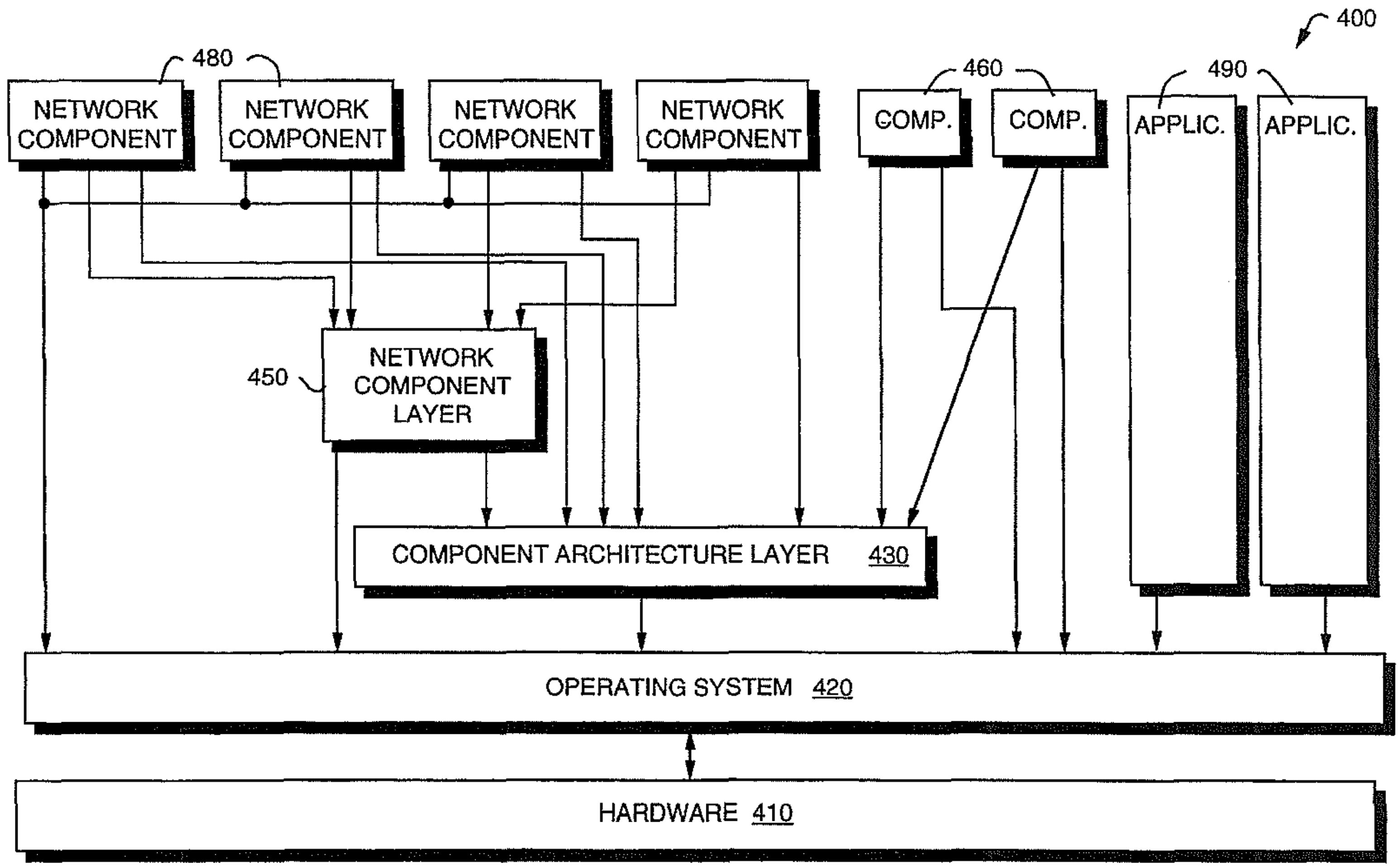

400
480
460
490
NETWORK COMPONENT
NETWORK COMPONENT
NETWORK COMPONENT
NETWORK COMPONENT
COMP.
COMP.
APPLIC.
APPLIC.
450
NETWORK COMPONENT LAYER
COMPONENT ARCHITECTURE LAYER 430
OPERATING SYSTEM 420
HARDWARE 410

FIG. 4

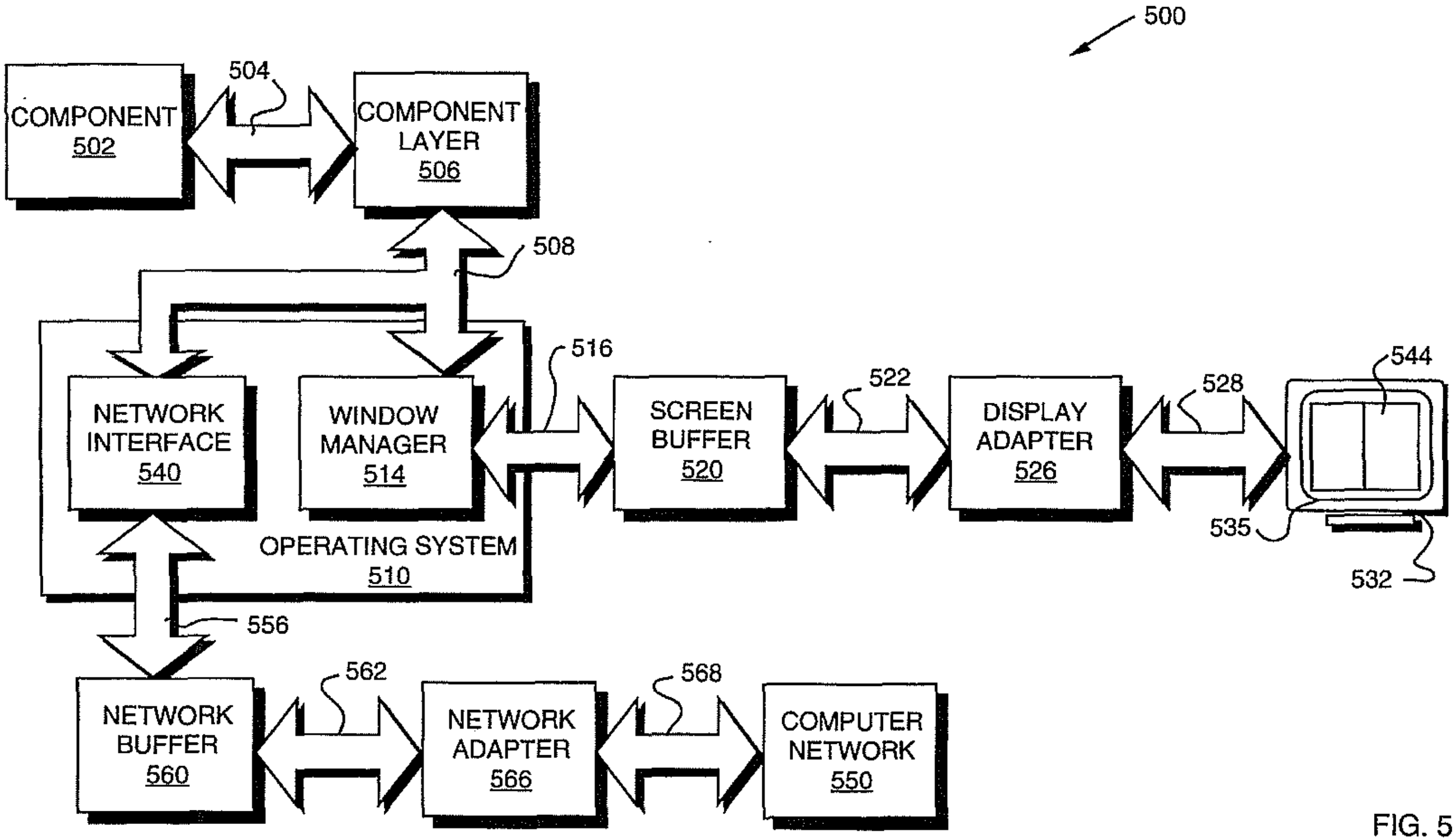

500
COMPONENT
502
504
COMPONENT
LAYER
506
508
NETWORK
INTERFACE
540
WINDOW
MANAGER
514
OPERATING SYSTEM
510
516
SCREEN
BUFFER
520
522
DISPLAY
ADAPTER
526
528
544
535
532
556
NETWORK
BUFFER
560
562
NETWORK
ADAPTER
566
568
COMPUTER
NETWORK
550

FIG. 5

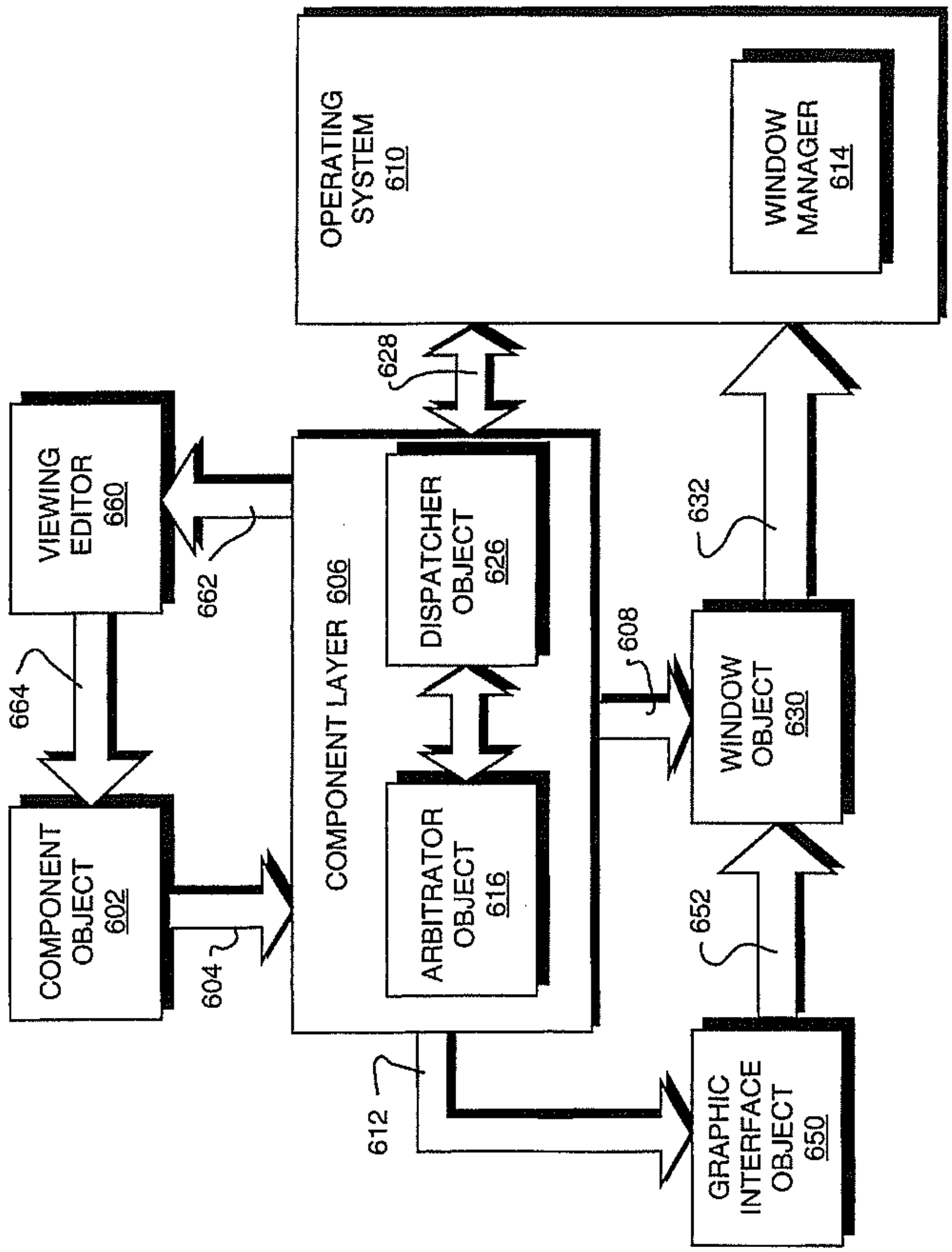

OPERATING SYSTEM 610
WINDOW MANAGER 614
628
VIEWING EDITOR 660
662
664
COMPONENT LAYER 606
DISPATCHER OBJECT 626
ARBITRATOR OBJECT 616
632
608
WINDOW OBJECT 630
COMPONENT OBJECT 602
604
652
612
GRAPHIC INTERFACE OBJECT 650

FIG. 6

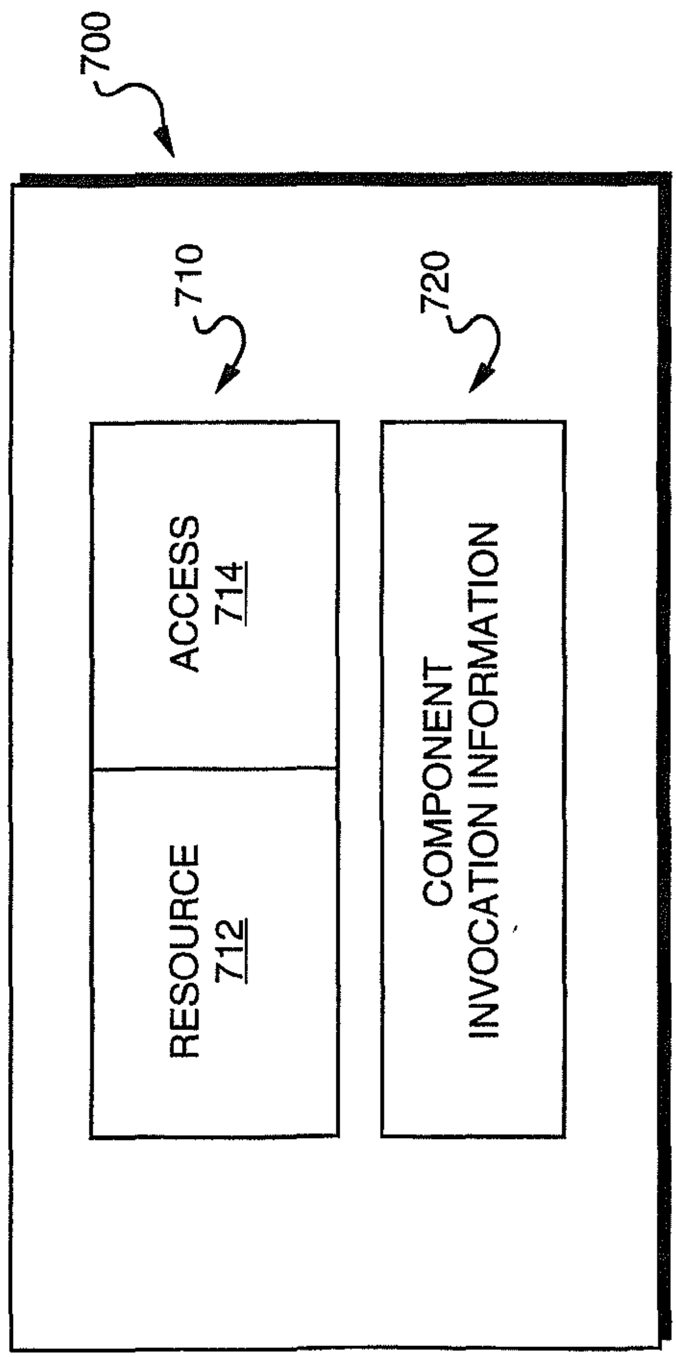

700
710
720
RESOURCE
712
ACCESS
714
COMPONENT
INVOCATION INFORMATION

FIG. 7

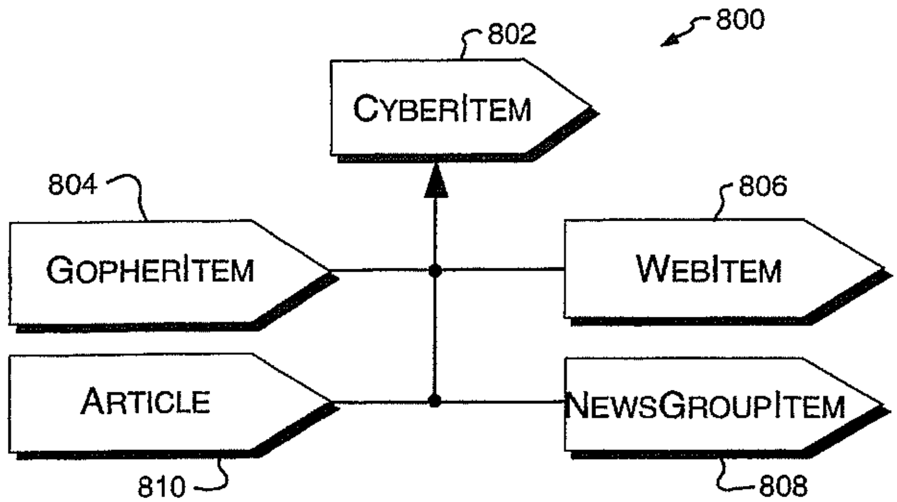

800
802
CYBERITEM
804
GOPHERITEM
806
WEBITEM
ARTICLE
NEWSGROUPITEM
810
808

FIG. 8

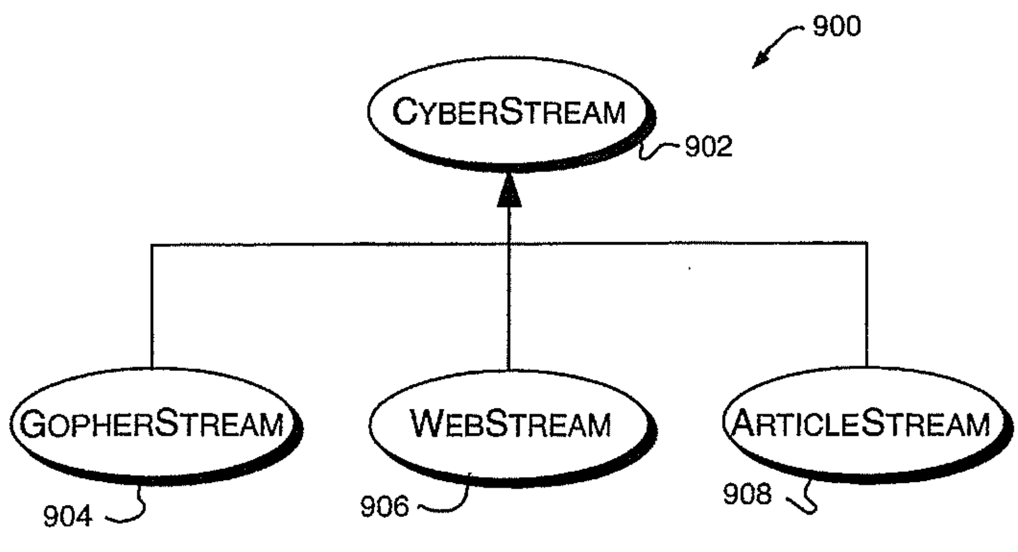

900
CYBERSTREAM
902
GOPHERSTREAM
WEBSTREAM
ARTICLESTREAM
904
906
908

FIG. 9

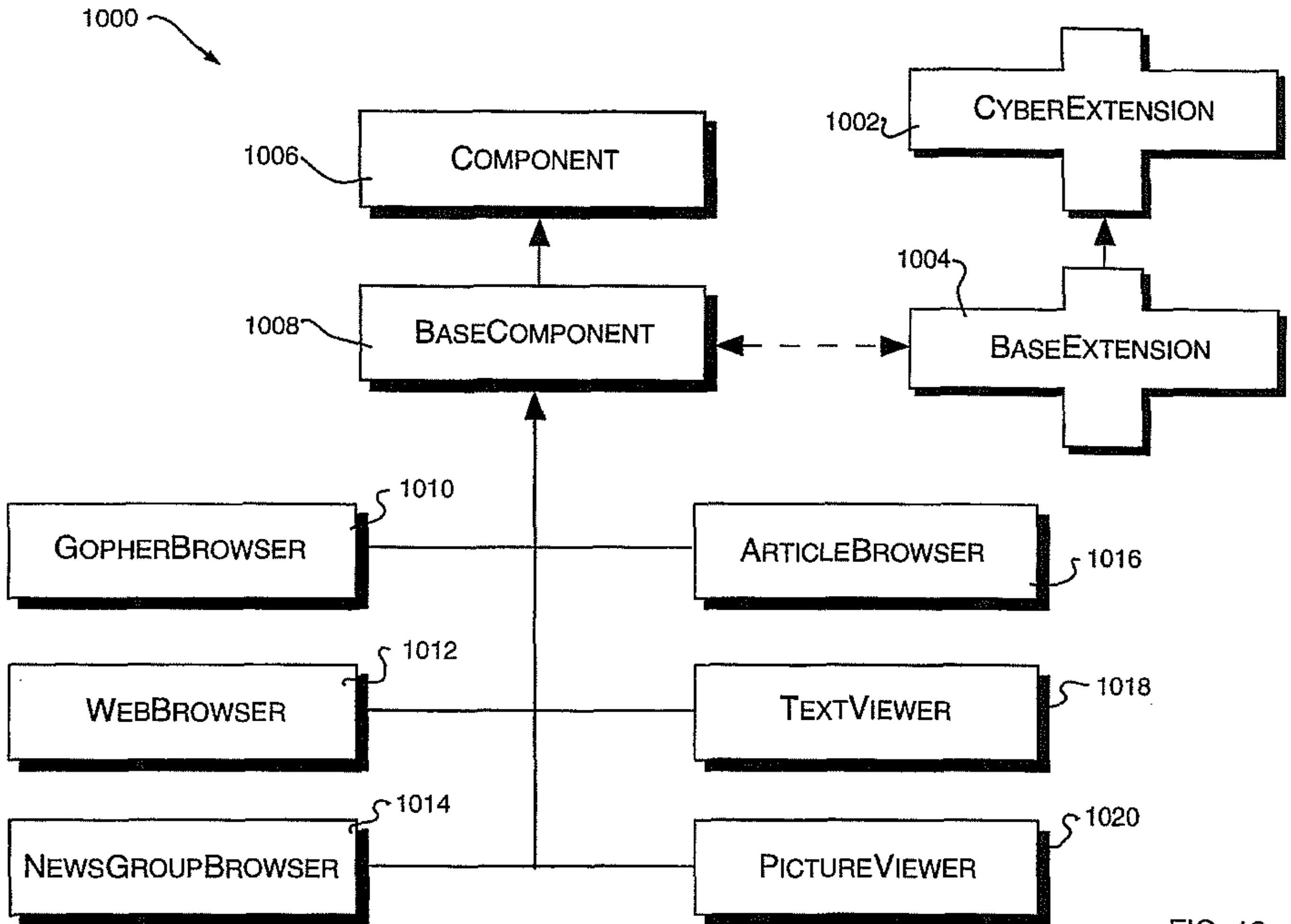
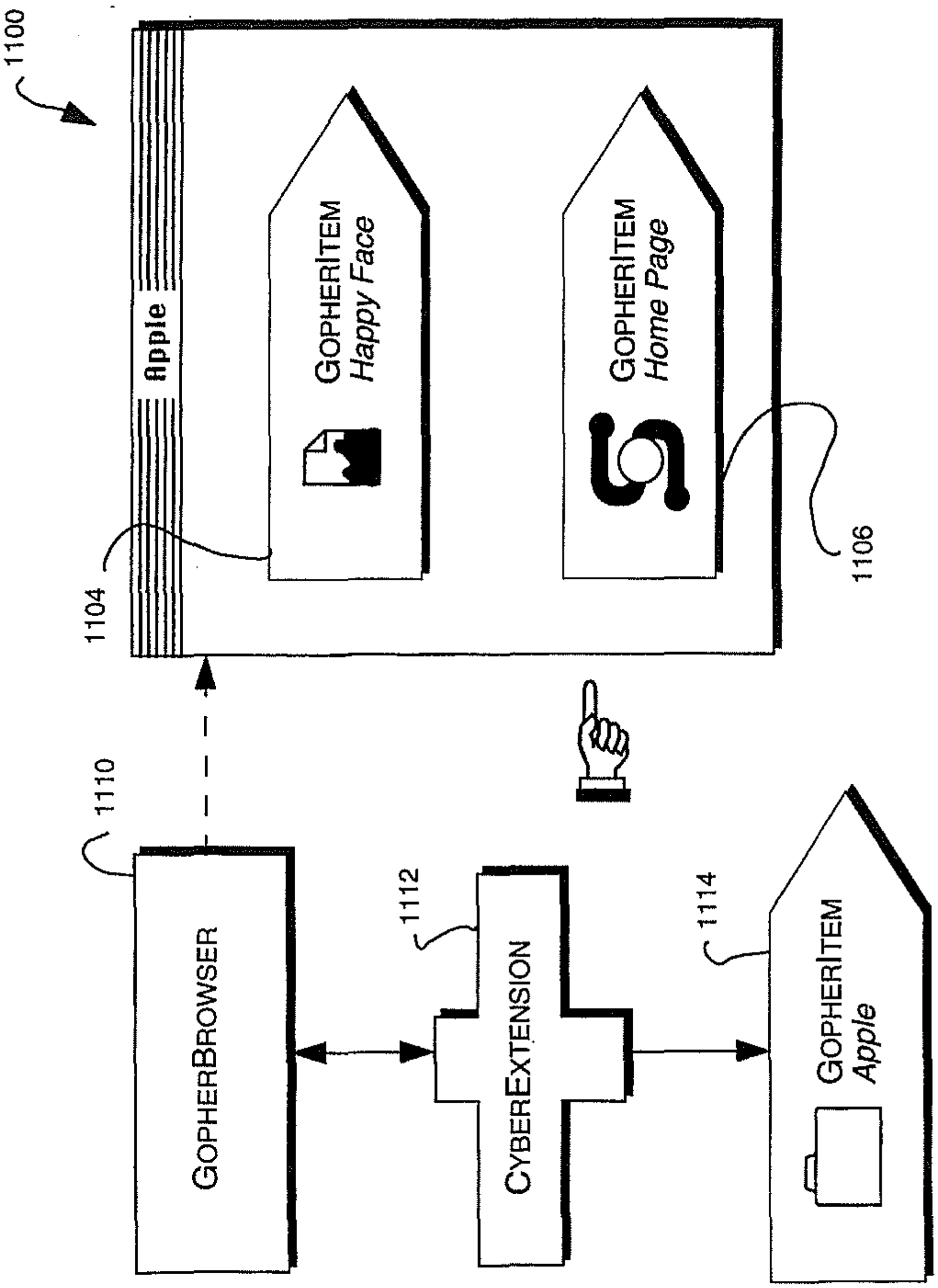

1000
1006
COMPONENT
1002
CYBEREXTENSION
1004
1008
BASECOMPONENT
BASEEXTENSION
1010
GOPHERBROWSER
ARTICLEBROWSER
1016
1012
WEBBROWSER
TEXTVIEWER
1018
1014
NEWSGROUPBROWSER
PICTUREVIEWER
1020

FIG. 10



1100
Apple
GopherItem
Happy Face
GopherItem
Home Page
1104
1106
1110
GopherBrowser
1112
CyberExtension
1114
GopherItem
Apple

FIG. 11A

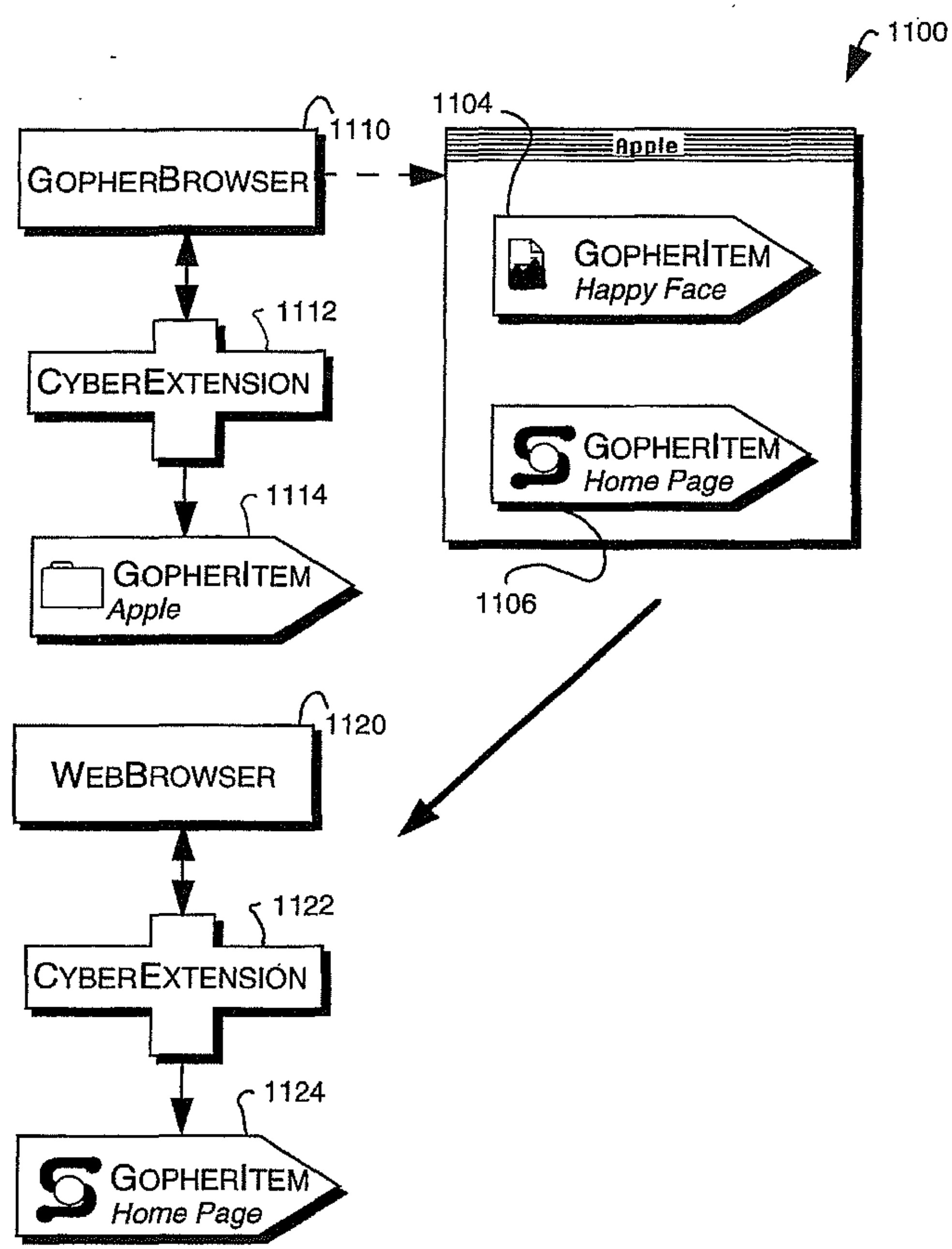

1100
1110
GOPHERBROWSER
1104
Apple
GOPHERITEM
Happy Face
1112
CYBEREXTENSION
GOPHERITEM
Home Page
1114
GOPHERITEM
Apple
1106
1120
WEBBROWSER
1122
CYBEREXTENSION
1124
GOPHERITEM
Home Page

FIG. 11B

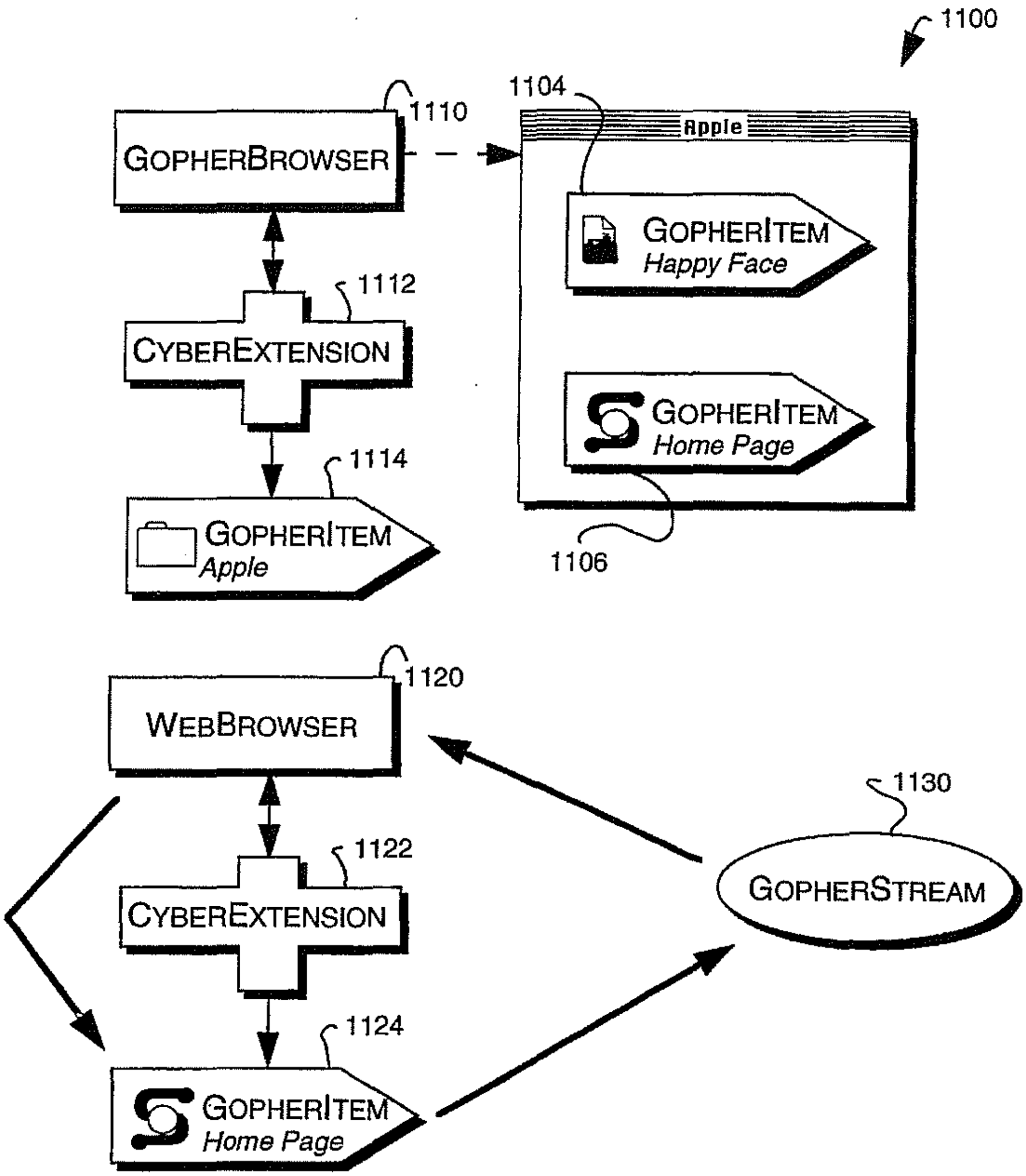

1100
1104
1110
GOPHERBROWSER
Apple
GOPHERITEM
Happy Face
1112
CYBEREXTENSION
GOPHERITEM
Home Page
1114
GOPHERITEM
Apple
1106
1120
WEBBROWSER
1130
1122
GOPHERSTREAM
CYBEREXTENSION
1124
GOPHERITEM
Home Page

FIG. 11C

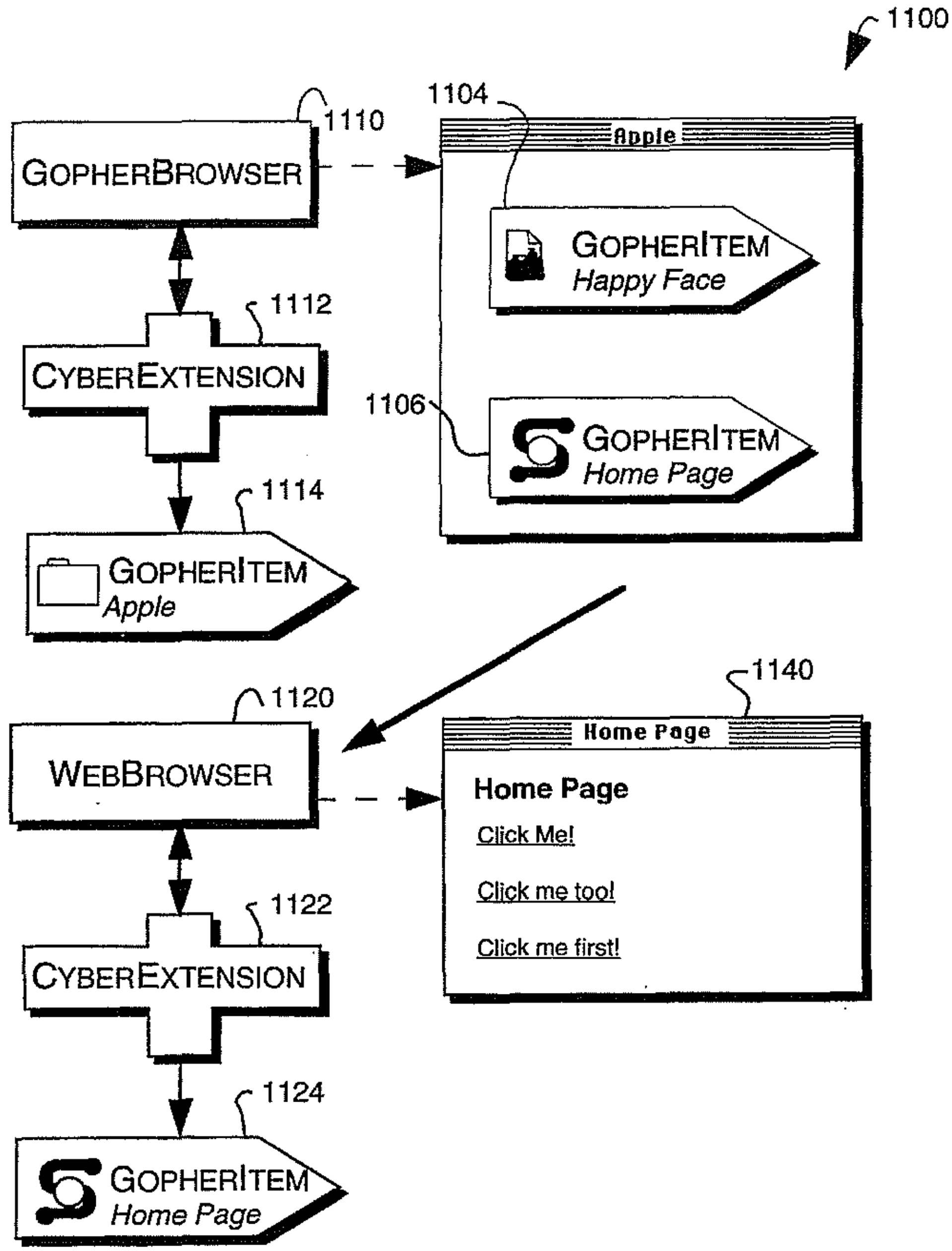

1100
1104
1110
Apple
GOPHERBROWSER
GOPHERITEM
Happy Face
1112
CYBEREXTENSION
1106
GOPHERITEM
Home Page
1114
GOPHERITEM
Apple
1140
1120
Home Page
WEBBROWSER
Home Page
Click Me!
Click me too!
Click me first!
1122
CYBEREXTENSION
1124
GOPHERITEM
Home Page

FIG. 11D

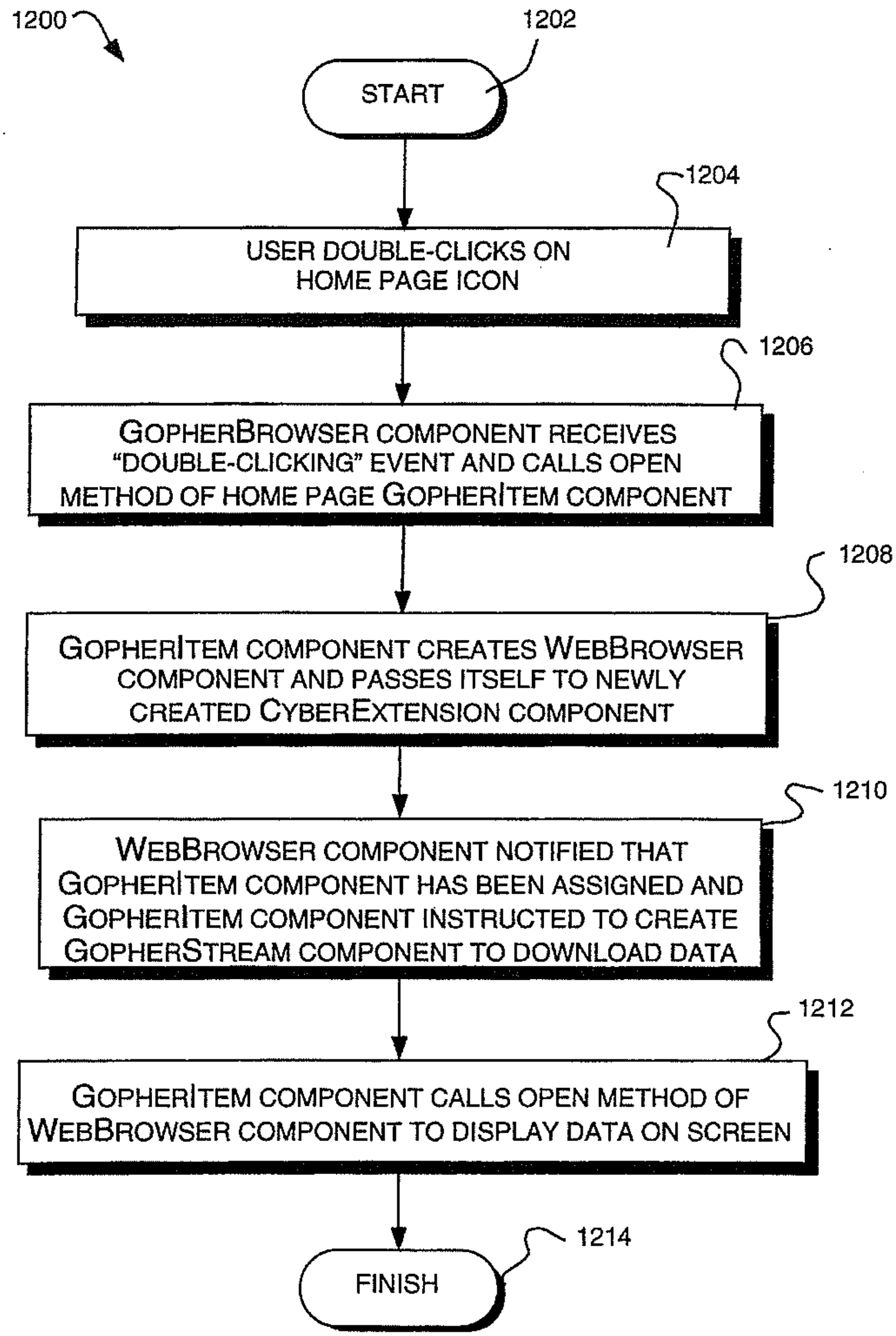

1200
1202
START
1204
USER DOUBLE-CLICKS ON HOME PAGE ICON
1206
GOPHERBROWSER COMPONENT RECEIVES "DOUBLE-CLICKING" EVENT AND CALLS OPEN METHOD OF HOME PAGE GOPHERITEM COMPONENT
1208
GOPHERITEM COMPONENT CREATES WEBBROWSER COMPONENT AND PASSES ITSELF TO NEWLY CREATED CYBEREXTENSION COMPONENT
1210
WEBBROWSER COMPONENT NOTIFIED THAT GOPHERITEM COMPONENT HAS BEEN ASSIGNED AND GOPHERITEM COMPONENT INSTRUCTED TO CREATE GOPHERSTREAM COMPONENT TO DOWNLOAD DATA
1212
GOPHERITEM COMPONENT CALLS OPEN METHOD OF WEBBROWSER COMPONENT TO DISPLAY DATA ON SCREEN
1214
FINISH

FIG. 12

ENCAPSULATED NETWORK ENTITY REFERENCE OF A NETWORK COMPONENT SYSTEM

This application is a continuation of U.S. patent application Ser. No. 08/435,880, filed May 5, 1995, now abandoned.

CROSS-REFERENCE TO RELATED APPLICATIONS

This invention is related to the following copending U.S. patent applications:

U.S. patent application Ser. No. 08/435,377, titled EXTENSIBLE, REPLACEABLE NETWORK COMPONENT SYSTEM;

U.S. Pat. No. 5,784,619 issued Jul. 21, 1998, titled REPLACEABLE AND EXTENSIBLE NOTEBOOK COMPONENT OF A NETWORK COMPONENT SYSTEM;

U.S. patent application Ser. No. 08/435,862, titled REPLACEABLE AND EXTENSIBLE LOG COMPONENT OF A NETWORK COMPONENT SYSTEM;

U.S. Pat. No. 5,724,506, issued Mar. 3, 1998, titled REPLACEABLE AND EXTENSIBLE CONNECTION DIALOG COMPONENT OF A NETWORK COMPONENT SYSTEM; and

U.S. Pat. No. 5,781,189 issued Jul. 14, 1998, titled EMBEDDING INTERNET BROWSER/BUTTONS WITHIN COMPONENTS OF A NETWORK COMPONENT SYSTEM, each of which was filed May 5, 1995 and assigned to the assignee of the present invention.

FIELD OF THE INVENTION

This invention relates generally to computer networks and, more particularly, to an architecture and tools for building Internet-specific services.

BACKGROUND OF THE INVENTION

The Internet is a system of geographically distributed computer networks interconnected by computers executing networking protocols that allow users to interact and share information over the networks. Because of such wide-spread information sharing, the Internet has generally evolved into an "open" system for which developers can design software for performing specialized operations, or services, essentially without restriction. These services are typically implemented in accordance with a client/server architecture, wherein the clients, e.g., personal computers or workstations, are responsible for interacting with the users and the servers are computers configured to perform the services as directed by the clients.

Not surprisingly, each of the services available over the Internet is generally defined by its own networking protocol. A protocol is a set of rules governing the format and meaning of messages or "packets" exchanged over the networks. By implementing services in accordance with the protocols, computers cooperate to perform various operations, or similar operations in various ways, for users wishing to "interact" with the networks. The services typically range from browsing or searching for information having a particular data format using a particular protocol to actually acquiring information of a different format in accordance with a different protocol.

For example, the file transfer protocol (FTP) service facilitates the transfer and sharing of files across the Internet.

The Telnet service allows users to log onto computers coupled to the networks, while the netnews protocol provides a bulletin-board service to its subscribers. Furthermore, the various data formats of the information available on the Internet include JPEG images, MPEG movies and μ-law sound files.

Two fashionable services for accessing information over the Internet are Gopher and the World-Wide Web ("Web"). Gopher consists of a series of Internet servers that provide a "list-oriented" interface to information available on the networks; the information is displayed as menu items in a hierarchical manner. Included in the hierarchy of menus are documents, which can be displayed or saved, and searchable indexes, which allow users to type keywords and perform searches.

Some of the menu items displayed by Gopher are links to information available on other servers located on the networks. In this case, the user is presented with a list of available information documents that can be opened. The opened documents may display additional lists or they may contain various data-types, such as pictures or text; occasionally, the opened documents may "transport" the user to another computer on the Internet.

The other popular information service on the Internet is the Web. Instead of providing a user with a hierarchical list-oriented view of information, the Web provides the user with a "linked-hypertext" view. Metaphorically, the Web perceives the Internet as a vast book of pages, each of which may contain pictures, text, sound, movies or various other types of data in the form of documents. Web documents are written in HyperText Markup Language (HTML) and Web servers transfer HTML documents to each other through the HyperText Transfer Protocol (HTTP).

The Web service is essentially a means for naming sources of information on the Internet. Armed with such a general naming convention that spans the entire network system, developers are able to build information servers that potentially any user can access. Accordingly, Gopher servers, HTTP servers, FTP servers, and E-mail servers have been developed for the Web. Moreover, the naming convention enables users to identify resources (such as documents) on any of these servers connected to the Internet and allow access to those resources.

As an example, a user "traverses" the Web by following hot items of a page displayed on a graphical Web browser. These hot items are hypertext links whose presence are indicated on the page by visual cues, e.g., underlined words, icons or buttons. When a user follows a link (usually by clicking on the cue with a mouse), the browser displays the target pointed to by the link which, in some cases, may be another HTML document.

The Gopher and Web information services represent entirely different approaches to interacting with information on the Internet. One follows a list-approach to information that "looks" like a telephone directory service, while the other assumes a page-approach analogous to a tabloid newspaper. However, both of these approaches include applications for enabling users to browse information available on Internet servers. Additionally, each of these applications has a unique way of viewing and accessing the information on the servers.

Netscape Navigator™ ("Netscape") is an example of a monolithic Web browser application that is configured to interact with many of the previously-described protocols, including HTTP, Gopher and FTP. When instructed to invoke an application that uses one of these protocols,

Netscape "translates" the protocol to hypertext. This translation places the user farther away from the protocol designed to run the application and, in some cases, actually thwarts the user's Internet experience. For example, a discussion system requiring an interactive exchange between participants may be bogged down by hypertext translations.

The Gopher and Web services may further require additional applications to perform specific functions, such as playing sound or viewing movies, with respect to the data types contained in the documents. For example, Netscape employs helper applications for executing applications having data formats it does not "understand". Execution of these functions on a computer requires interruption of processing and context switching (i.e., saving of state) prior to invoking the appropriate application. Thus, if a user operating within the Netscape application "opens" a MPEG movie, that browsing application must be saved (e.g., to disk) prior to opening an appropriate MPEG application, e.g., Sparkle, to view the image. Such an arrangement is inefficient and rather disruptive to processing operations of the computer.

Typically, a computer includes an operating system and application software which, collectively, control the operations of the computer. The applications are preferably task-specific and independent, e.g., a word processor application edits words, a drawing application edits drawings and a database application interacts with information stored on a database storage unit. Although a user can move data from one application to the other, such as by copying a drawing into a word processing file, the independent applications must be invoked to thereafter manipulate that data.

Generally, the application program presents information to a user through a window of a graphical user interface by drawing images, graphics or text within the window region. The user, in turn, communicates with the application by "pointing" at graphical objects in the window with a pointer that is controlled by a hand-operated pointing device, such as a mouse, or by pressing keys of a keyboard.

The graphical objects typically included with each window region are sizing boxes, buttons and scroll bars. These objects represent user interface elements that the user can point at with the pointer (or a cursor) to select or manipulate. For example, the user may manipulate these elements to move the windows around on the display screen, and change their sizes and appearances so as to arrange the window in a convenient manner. When the elements are selected or manipulated, the underlying application program is informed, via the window environment, that control has been appropriated by the user.

A menu bar is a further example of a user interface element that provides a list of menus available to a user. Each menu, in turn, provides a list of command options that can be selected merely by pointing to them with the mouse-controlled pointer. That is, the commands may be issued by actuating the mouse to move the pointer onto or near the command selection, and pressing and quickly releasing, i.e., "clicking" a button on the mouse.

In contrast to this typical application-based computing environment, a software component architecture provides a modular document-based computing arrangement using tools such as viewing editors. The key to document-based computing is the compound document, i.e., a document composed of many different types of data sharing the same file. The types of data contained in a compound document may range from text, tables and graphics to video and sound. Several editors, each designed to handle a particular data type or format, can work on the contents of the document at the same time, unlike the application-based computing environment.

Since many editors may work together on the same document, the compound document is apportioned into individual modules of content for manipulation by the editors. The compound-nature of the document is realized by embedding these modules within each other to create a document having a mixture of data types. The software component architecture provides the foundation for assembling documents of differing contents and the present invention is directed to a system for extending this capability to network-oriented services.

To remotely access information stored on a resource of the network, the user typically invokes a service configured to operate in accordance with a protocol for accessing the resource. In particular, the user types an explicit destination address command that includes a uniform resource locator (URL). The URL is a rather long (approximately 50 character) address pointer that identifies both a network resource and a means for accessing that resource. The following is an example of a hypothetical URL address pointer to a remote resource on a Web server:

http:/aaaa.bbb.cc/hypertext/DdddEeeee/WWW/Ffffff.html

It is apparent that having to type such long destination address pointers can become quite burdensome for users that frequently access information from remote resources.

Therefore, it is among the objects of the present invention to simplify a user's experience on computer networks without sacrificing the flexibility afforded the user by employing existing protocols and data types available on those networks.

Another object of the invention is to provide a system for users to search and access information on the Internet without extensive understanding or knowledge of the underlying protocols and data formats needed to access that information.

Still another object of the invention is to provide users with a simple means for remotely accessing information stored on resources connected to computer networks.

SUMMARY OF THE INVENTION

Briefly, the invention comprises a network-oriented component system for efficiently accessing information from a network resource located on a computer network by creating an encapsulated network entity that contains a reference to that resource. The encapsulated entity is preferably implemented as a network component stored on a computer remotely displaced from the referenced resource. In addition, the encapsulated entity may be manifested as a visual object on a graphical user interface of a computer screen. Such visual manifestation allows a user to easily manipulate the entity in order to display the contents of the resource on the screen or to electronically forward the entity over the network.

In the illustrative embodiment of the invention, the reference to the network resource is preferably a "pointer", such as a uniform resource locator (URL), that identifies the network address of the resource, e.g., a Gopher browser or a Web page. In addition to storing the pointer, the encapsulated entity also contains information for invoking appropriate network components needed to access the resource. Communication among the network components is achieved through novel application programming interfaces (APIs) to facilitate integration with an underlying software component architecture. Such a cooperating architecture allows the encapsulated entity and network components to "transport" the user to the network location of the remote resource.

Specifically, the encapsulated entity component is an object of the network-oriented component system that is preferably embodied as a customized framework having a set of interconnected abstract classes. A CyberItem class defines the encapsulated entity object which interacts with other objects of the network system to remotely access information from the referenced resource. Since these objects are integral elements of the cooperating component architecture, any type of encapsulated network entity may be developed with consistent behaviors, i.e., these entities may be manifested as visual objects that can be distributed and manipulated iconically.

Advantageously, the inventive encapsulation technique described herein provides a user with a simple means for accessing information on computer networks.

BRIEF DESCRIPTION OF THE DRAWINGS

The above and further advantages of the invention may be better understood by referring to the following description in conjunction with the accompanying drawings in which:

FIG. 1 is a block diagram of a network system including a collection of computer networks interconnected by client and server computers;

FIG. 2 is a block diagram of a client computer, such as a personal computer, on which the invention may advantageously operate;

FIG. 3 is a block diagram of the server computer of FIG. 1;

FIG. 4 is a highly schematized block diagram of a layered component computing arrangement in accordance with the invention;

FIG. 5 is a schematic illustration software of the interaction of a component, a software component layer and an operating system of the computer of FIG. 2;

FIG. 6 is a schematic illustration of the interaction between a component, a component layer and a window manager in accordance with the invention;

FIG. 7 is a schematic diagram of an illustrative encapsulated network entity object in accordance with the invention;

FIG. 8 is a simplified class heirarchy diagram illustrating a base class CyberItem, and its associated subclasses, used to construct network component objects in accordance with the invention;

FIG. 9 is a simplified class heirarchy diagram illustrating a base class CyberStream, and its associated subclasses, in accordance with the invention;

FIG. 10 is a simplified class hierarchy diagram illustrating a base class CyberExtension, and its associated subclasses, in accordance with the present invention;

FIGS. 11A–11D are highly schematized diagrams illustrating the interactions between the network component objects, including the encapsulated network entity object of FIG. 7; and

FIG. 12 is an illustrative flowchart of the sequence of steps involved in invoking, and accessing, information from a referenced network resource.

DETAILED DESCRIPTION OF ILLUSTRATIVE EMBODIMENT

FIG. 1 is a block diagram of a network system 100 comprising a collection of computer networks 110 interconnected by client computers ("clients") 200, e.g., workstations or personal computers, and server computers ("servers") 300. The servers are typically computers having

hardware and software elements that provide resources or services for use by the clients 200 to increase the efficiency of their operations. It will be understood to those skilled in the art that, in an alternate embodiment, the client and server may exist on the same computer; however, for the illustrative embodiment described herein, the client and server are separate computers.

Several types of computer networks 110, including local area networks (LANs) and wide area networks (WANs), may be employed in the system 100. A LAN is a limited area network that typically consists of a transmission medium, such as coaxial cable or twisted pair, while a WAN may be a public or private telecommunications facility that interconnects computers widely dispersed. In the illustrative embodiment, the network system 100 is the Internet system of geographically distributed computer networks.

Computers coupled to the Internet typically communicate by exchanging discrete packets of information according to predefined networking protocols. Execution of these networking protocols allow users to interact and share information across the networks. As an illustration, in response to a user's request for a particular service, the client 200 sends an appropriate information packet to the server 300, which performs the service and returns a result back to the client 200.

FIG. 2 illustrates a typical hardware configuration of a client 200 comprising a central processing unit (CPU) 210 coupled between a memory 214 and input/output (I/O) circuitry 218 by bidirectional buses 212 and 216. The memory 214 typically comprises random access memory (RAM) for temporary storage of information and read only memory (ROM) for permanent storage of the computer's configuration and basic operating commands, such as portions of an operating system (not shown). As described further herein, the operating system controls the operations of the CPU 210 and client computer 200.

The I/O circuitry 218, in turn, connects the computer to computer networks, such as the Internet networks 250, via a bidirectional bus 222 and to cursor/pointer control devices, such as a keyboard 224 (via cable 226) and a mouse 230 (via cable 228). The mouse 230 typically contains at least one button 234 operated by a user of the computer. A conventional display monitor 232 having a display screen 235 is also connected to I/O circuitry 218 via cable 238. A pointer (cursor) 240 is displayed on windows 244 of the screen 235 and its position is controllable via the mouse 230 or the keyboard 224, as is well-known. The I/O circuitry 218 receives information, such as control and data signals, from the mouse 230 and keyboard 224, and provides that information to the CPU 210 for display on the screen 235 or, as described further herein, for transfer over the Internet 250.

FIG. 3 illustrates a typical hardware configuration of a server 300 of the network system 100. The server 300 has many of the same units as employed in the client 200, including a CPU 310, a memory 314 and I/O circuitry 318, each of which are interconnected by bidirectional buses 312 and 316. Also, the I/O circuitry connects the computer to computer networks 350 via a bidirectional bus 322. These units are configured to perform functions similar to those provided by their corresponding units in the computer 200. In addition, the server typically includes a mass storage unit 320, such as a disk drive, connected to the I/O circuitry 318 via bidirectional bus 324.

It is to be understood that the I/O circuits within the computers 200 and 300 contain the necessary hardware, e.g., buffers and adapters, needed to interface with the control

devices, the display monitor, the mass storage unit and the network. Moreover, the operating system includes the necessary software drivers to control, e.g., network adapters within the I/O circuits when performing I/O operations, such as the transfer of data packets between the client 200 and server 300.

The computers are preferably personal computers of the Macintosh® series of computers sold by Apple Computer Inc., although the invention may also be practiced in the context of other types of computers, including the IBM® series of computers sold by International Business Machines Corp. These computers have resident thereon, and are controlled and coordinated by, operating system software, such as the Apple® System 7®, IBM OS2®, or the Microsoft® Windows® operating systems.

As noted, the present invention is based on a modular document computing arrangement as provided by an underlying software component architecture, rather than the typical application-based environment of prior computing systems. FIG. 4 is a highly schematized diagram of the hardware and software elements of a layered component computing arrangement 400 that includes the novel network-oriented component system of the invention. At the lowest level there is the computer hardware, shown as layer 410. Interfacing with the hardware is a conventional operating system layer 420 that includes a window manager, a graphic system, a file system and network-specific interfacing, such as a TCP/IP protocol stack and an Appletalk protocol stack.

The software component architecture is preferably implemented as a component architecture layer 430. Although it is shown as overlaying the operating system 420, the component architecture layer 430 is actually independent of the operating system and, more precisely, resides side-by-side with the operating system. This relationship allows the component architecture to exist on multiple platforms that employ different operating systems.

In accordance with the present invention, a novel network-oriented component layer 450 contains the underlying technology for creating encapsulated entity components that contain references to network resources located on computer networks. As described further herein, communication among these components is achieved through novel application programming interfaces (APIs) to ensure integration with the underlying component architecture layer 430. These novel APIs are preferably delivered in the form of objects in a class hierarchy.

It should be noted that the network component layer 450 may operate with any existing system-wide component architecture, such as the Object Linking and Embedding (OLE) architecture developed by the Microsoft Corporation; however, in the illustrative embodiment, the component architecture is preferably OpenDoc, the vendor-neutral, open standard for compound documents developed by, among others, Apple Computer, Inc.

Using tools such as viewing editors, the component architecture layer 430 creates a compound document composed of data having different types and formats. Each differing data type and format is contained in a fundamental unit called a computing part or, more generally, a "component" 460 comprised of a viewing editor along with the data content. An example of the computing component 460 may include a MacDraw component. The editor, on the other hand, is analogous to an application program in a conventional computer. That is, the editor is a software component which provides the necessary functionality to display a

component's contents and, where appropriate, present a user interface for modifying those contents. Additionally, the editor may include menus, controls and other user interface elements. The network component layer 450 extends the functionality of the underlying component architecture layer 430 by defining network-oriented components 480 that seamlessly integrate with these components 460 to provide basic tools for efficiently accessing information from network resources located on, e.g., servers coupled to the computer networks.

FIG. 4 also illustrates the relationship of applications 490 to the elements of the document computing arrangement 400. Although they reside in the same "user space" as the components 460 and network components 480, the applications 490 do not interact with these elements and, thus, interface directly to the operating system layer 420. Because they are designed as monolithic, autonomous modules, applications (such as previous Internet browsers) often do not even interact among themselves. In contrast, the components of the arrangement 400 are designed to work together and communicate via the common component architecture layer 430 or, in the case of the network components, via the novel network component layer 450.

Specifically, the invention features the provision of the network-oriented component system which, when invoked, causes actions to take place that enhance the ability of a user to interact with the computer to create encapsulated entities that contain references to network resources located on computer networks, such as the Internet. The encapsulated entities are manifested as visual objects to a user via a window environment, such as the graphical user interface provided by System 7 or Windows, that is preferably displayed on the screen 235 (FIG. 2) as a graphical display to facilitate interactions between the user and the computer, such as the client 200. This behavior of the system is brought about by the interaction of the network components with a series of system software routines associated with the operating system 420. These system routines, in turn, interact with the component architecture layer 430 to create the windows and graphical user interface elements, as described further herein.

The window environment is generally part of the operating system software 420 that includes a collection of utility programs for controlling the operation of the computer 200. The operating system, in turn, interacts with the components to provide higher level functionality, including a direct interface with the user. A component makes use of operating system functions by issuing a series of task commands to the operating system via the network component layer 450 or, as is typically the case, through the component architecture layer 430. The operating system 420 then performs the requested task. For example, the component may request that a software driver of the operating system initiate transfer of a data packet over the networks 250 or that the operating system display certain information on a window for presentation to the user.

FIG. 5 is a schematic illustration of the interaction of a component 502, software component layer 506 and an operating system 510 of a computer 500, which is similar to, and has equivalent elements of, the client computer 200 of FIG. 2. As noted, the network component layer 450 (FIG. 4) is integrated with the component architecture layer 430 to provide a cooperating architecture that allows any encapsulated entity and network component to "transport" the user to the network location of a remote resource; accordingly, for purposes of the present discussion, the layers 430 and 450 may be treated as a single software component layer 506.

The component 502, component layer 506 and operating system 510 interact to control and coordinate the operations of the computer 500 and their interaction is illustrated schematically by arrows 504 and 508. In order to display information on a screen display 535, the component 502 and component layer 506 cooperate to generate and send display commands to a window manager 514 of the operating system 510. The window manager 514 stores information directly (via arrow 516) into a screen buffer 520.

The window manager 514 is a system software routine that is generally responsible for managing windows 544 that the user views during operation of the network component system. That is, it is generally the task of the window manager to keep track of the location and size of the window and window areas which must be drawn and redrawn in connection with the network component system of the present invention.

Under control of various hardware and software in the system, the contents of the screen buffer 520 are read out of the buffer and provided, as indicated schematically by arrow 522, to a display adapter 526. The display adapter contains hardware and software (sometimes in the form of firmware) which converts the information in the screen buffer 520 to a form which can be used to drive a display screen 535 of a monitor 532. The monitor 532 is connected to display adapter 526 by cable 528.

Similarly, in order to transfer information as a packet over the computer networks, the component 502 and component layer 506 cooperate to generate and send network commands, such as remote procedure calls, to a network-specific interface 540 of the operating system 510. The network interface comprises system software routines, such as "stub" procedure software and protocol stacks, that are generally responsible for formating the information into a predetermined packet format according to the specific network protocol used, e.g., TCP/IP or Apple-talk protocol.

Specifically, the network interface 540 stores the packet directly (via arrow 556) into a network buffer 560. Under control of the hardware and software in the system, the contents of the network buffer 560 are provided, as indicated schematically by arrow 562, to a network adapter 566. The network adapter incorporates the software and hardware, i.e., electrical and mechanical interchange circuits and characteristics, needed to interface with the particular computer networks 550. The adapter 566 is connected to the computer networks 550 by cable 568.

In a preferred embodiment, the invention described herein is implemented in an object-oriented programming (OOP) language, such as C++, using System Object Model (SOM) technology and OOP techniques.

The C++ and SOM languages are well-known and many articles and texts are available which describe the languages in detail. In addition, C++ and SOM compilers are commercially available from several vendors. Accordingly, for reasons of brevity, the details of the C++ and SOM languages and the operations of their compilers will not be discussed further in detail herein.

As will be understood by those skilled in the art, OOP techniques involve the definition, creation, use and destruction of "objects". These objects are software entities comprising data elements and routines, or functions, which manipulate the data elements. The data and related functions are treated by the software as an entity that can be created, used and deleted as if it were a single item. Together, the data and functions enable objects to model virtually any real-world entity in terms of its characteristics, which can be represented by the data elements, and its behavior, which can be represented by its data manipulation functions. In this way, objects can model concrete things like computers, while also modeling abstract concepts like numbers or geometrical designs.

Objects are defined by creating "classes" which are not objects themselves, but which act as templates that instruct the compiler how to construct an actual object. A class may, for example, specify the number and type of data variables and the steps involved in the functions which manipulate the data. An object is actually created in the program by means of a special function called a "constructor" which uses the corresponding class definition and additional information, such as arguments provided during object creation, to construct the object. Likewise objects are destroyed by a special function called a "destructor". Objects may be used by manipulating their data and invoking their functions.

The principle benefits of OOP techniques arise out of three basic principles: encapsulation, polymorphism and inheritance. Specifically, objects can be designed to hide, or encapsulate, all, or a portion of, its internal data structure and internal functions. More specifically, during program design, a program developer can define objects in which all or some of the data variables and all or some of the related functions are considered "private" or for use only by the object itself. Other data or functions can be declared "public" or available for use by other programs. Access to the private variables by other programs can be controlled by defining public functions for an object which access the object's private data. The public functions form a controlled and consistent interface between the private data and the "outside" world. Any attempt to write program code which directly accesses the private variables causes the compiler to generate an error during program compilation which error stops the compilation process and prevents the program from being run.

Polymorphism is a concept which allows objects and functions that have the same overall format, but that work with different data, to function differently in order to produce consistent results. Inheritance, on the other hand, allows program developers to easily reuse pre-existing programs and to avoid creating software from scratch. The principle of inheritance allows a software developer to declare classes (and the objects which are later created from them) as related. Specifically, classes may be designated as subclasses of other base classes. A subclass "inherits" and has access to all of the public functions of its base classes just as if these functions appeared in the subclass. Alternatively, a subclass can override some or all of its inherited functions or may modify some or all of its inherited functions merely by defining a new function with the same form (overriding or modification does not alter the function in the base class, but merely modifies the use of the function in the subclass). The creation of a new subclass which has some of the functionality (with selective modification) of another class allows software developers to easily customize existing code to meet their particular needs.

In accordance with the present invention, the component 502 and windows 544 are "objects" created by the component layer 506 and the window manager 514, respectively, the latter of which may be an object-oriented program. Interaction between a component, component layer and a window manager is illustrated in greater detail in FIG. 6.

In general, the component layer 606 interfaces with the window manager 614 by creating and manipulating objects. The window manager itself may be an object which is

created when the operating system is started. Specifically, the component layer creates window objects 630 that cause the window manager to create associated windows on the display screen. This is shown schematically by an arrow 608. In addition, the component layer 606 creates individual graphic interface objects 650 that are stored in each window object 630, as shown schematically by arrows 612 and 652. Since many graphic interface objects may be created in order to display many interface elements on the display screen, the window object 630 communicates with the window manager by means of a sequence of drawing commands issued from the window object to the window manager 614, as illustrated by arrow 632.

As noted, the component layer 606 functions to embed components within one another to form a compound document having mixed data types and formats. Many different viewing editors may work together to display, or modify, the data contents of the document. In order to direct keystrokes and mouse events initiated by a user to the proper components and editors, the component layer 606 includes an arbitrator 616 and a dispatcher 626.

The dispatcher is an object that communicates with the operating system 610 to identify the correct viewing editor 660, while the arbitrator is an object that informs the dispatcher as to which editor "owns" the stream of keystrokes or mouse events. Specifically, the dispatcher 626 receives these "human-interface" events from the operating system 610 (as shown schematically by arrow 628) and delivers them to the correct viewing editor 660 via arrow 662. The viewing editor 660 then modifies or displays, either visually or acoustically, the contents of the data types.

Although OOP offers significant improvements over other programming concepts, software development still requires significant outlays of time and effort, especially if no pre-existing software is available for modification. Consequently, a prior art approach has been to provide a developer with a set of predefined, interconnected classes which create a set of objects and additional miscellaneous routines that are all directed to performing commonly-encountered tasks in a particular environment. Such predefined classes and libraries are typically called "frameworks" and essentially provide a pre-fabricated structure for a working document.

For example, a framework for a user interface might provide a set of predefined graphic interface objects which create windows, scroll bars, menus, etc. and provide the support and "default" behavior for these interface objects. Since frameworks are based on object-oriented techniques, the predefined classes can be used as base classes and the built-in default behavior can be inherited by developer-defined subclasses and either modified or overridden to allow developers to extend the framework and create customized solutions in a particular area of expertise. This object-oriented approach provides a major advantage over traditional programming since the programmer is not changing the original program, but rather extending the capabilities of that original program. In addition, developers are not blindly working through layers of code because the framework provides architectural guidance and modeling and, at the same time, frees the developers to supply specific actions unique to the problem domain.

There are many kinds of frameworks available, depending on the level of the system involved and the kind of problem to be solved. The types of frameworks range from high-level frameworks that assist in developing a user interface, to lower-level frameworks that provide basic system software services such as communications, printing, file systems support, graphics, etc. Commercial examples of application-type frameworks include MacApp (Apple), Bedrock (Symantec), OWL (Borland), NeXT Step App Kit (NeXT) and Smalltalk-80 MVC (ParcPlace).

While the framework approach utilizes all the principles of encapsulation, polymorphism, and inheritance in the object layer, and is a substantial improvement over other programming techniques, there are difficulties which arise. These difficulties are caused by the fact that it is easy for developers to reuse their own objects, but it is difficult for the developers to use objects generated by other programs. Further, frameworks generally consist of one or more object "layers" on top of a monolithic operating system and even with the flexibility of the object layer, it is still often necessary to directly interact with the underlying system by means of awkward procedure calls.

In the same way that a framework provides the developer with prefab functionality for a document, a system framework, such as that included in the preferred embodiment, can provide a prefab functionality for system level services which developers can modify or override to create customized solutions, thereby avoiding the awkward procedural calls necessary with the prior art frameworks. For example, consider a customizable network interface framework which can provide the foundation for browsing and accessing information over a computer network. A software developer who needed these capabilities would ordinarily have to write specific routines to provide them. To do this with a framework, the developer only needs to supply the characteristic and behavior of the finished output, while the framework provides the actual routines which perform the tasks.

A preferred embodiment takes the concept of frameworks and applies it throughout the entire system, including the document, component, component layer and the operating system. For the commercial or corporate developer, systems integrator, or OEM, this means all of the advantages that have been illustrated for a framework, such as MacApp, can be leveraged not only at the application level for things such as text and graphical user interfaces, but also at the system level for such services as printing, graphics, multi-media, file systems and, as described herein, network-specific operations.

Referring again to FIG. 6, the window object 630 and the graphic interface object 650 are elements of a graphical user interface of a network component system having a customizable framework for greatly enhancing the ability of a user to efficiently access information from a network resource on computer networks by creating an encapsulated entity that contains a reference to that resource. The encapsulated entity is preferably implemented as a network component of the system and stored as a visual object, e.g., an icon, for display on a graphical user interface. Such visual display allows a user to easily manipulate the entity component to display the contents of the resource on a computer screen or to electronically forward the entity over the networks.

Furthermore, the reference to the network resource is a pointer that identifies the network address of the resource, e.g., a Gopher browser, a Web page or an E-mail message. FIG. 7 is a schematic diagram of an illustrative encapsulated network entity object 700 containing a pointer 710. In one embodiment of the invention, the pointer may be a uniform resource locator (URL) having a first portion 712 that identifies the particular network resource and a second portion 714 that specifies the means for accessing that

resource. More specifically, the URL is a string of approximately 50 characters that describes the protocol used to address the target resource, the server on which the resource resides, the path to the resource and the resource filename. It is to be understood, however, that other representations of a "pointer" are included within the principles of the invention, e.g., a Post Office Protocol (POP) account and message identification (ID).

In addition to storing the pointer, the encapsulated entity also contains information **720** for invoking appropriate network components needed to access the resource. Communication among these network components is achieved through novel application programming interfaces (APIs). These APIs are preferably delivered in the form of objects in a class hierarchy that is extensible so that developers can create new components. From an implementation viewpoint, the objects can be subclassed and can inherit from base classes to build customized components that allow users to see different kinds of data using different kinds of protocols, or to create components that function differently from existing components.

In accordance with the invention, the customized framework has a set of interconnected abstract classes for defining network-oriented objects used to build the customized network components. These abstract classes include CyberItem, CyberStream and CyberExtension and the objects they define are used to build the novel network components. A description of these abstract classes is provided in copending and commonly assigned U.S. patent application titled Extensible, Replaceable Network Component System, filed May 5, 1995, which application is incorporated by reference as though fully set forth herein.

Specifically, the CyberItem class defines the encapsulated entity object which interacts with objects defined by the other abstract classes of the network system to "transport" the user to the network location, i.e., remotely access information from the referenced resource and display that information to the user at the computer. Since these objects are integral elements of the cooperating component architecture, any type of encapsulated network entity may be developed with consistent behaviors, i.e., these entities may be manifested as visual objects that can be distributed and manipulated iconically.

FIG. **8** illustrates a simplified class hierarchy diagram **800** of the base class CyberItem **802** used to construct the encapsulated network entity component object **602**. In accordance with the illustrative embodiment, subclasses of the CyberItem base class are used to construct various network component objects configured to provide such services for the novel network-oriented component system. For example, the subclass GopherItem **804** is derived from the CyberItem class **802** and encapsulates a network entity component object representing a "thing in Gopher space", such as a Gopher directory.

Since each of the classes used to construct these network component objects are subclasses of the CyberItem base class, each class inherits the functional operators and methods that are available from that base class. Accordingly, methods associated with the CyberItem base class for, e.g., instructing an object to open itself, are assumed by the subclasses to allow the network components to display CyberItem objects in a consistent manner.

In some instances, a CyberItem object may need to spawn a CyberStream object in order to obtain the actual data for the object it represents. FIG. 9 illustrates a simplified class hierarchy diagram **900** of the base class CyberStream **902**

which is an abstraction that serves as an API between a component configured to display a particular data format and the method for obtaining the actual data. Specifically, a CyberStream object contains the software commands necessary to create a "data stream" for transfering information from one object to another. According to the invention, a GopherStream subclass **904** is derived from the CyberStream base class **902** and encapsulates a network object that implements the Gopher protocol.

FIG. **10** is a simplified class hierarchy diagram **1000** of the base class CyberExtension **1002** which represents additional behaviors provided to components of the underlying software component architecture. For example, CyberExtension objects add functionality to, and extend the APIs of, existing components so that they may communicate with the novel network components, such as the encapsulated entity objects. As a result, the CyberExtension base class **1002** operates in connection with a Component base class **1006** through their respective subclasses BaseExtension **1004** and BaseComponent **1008**.

CyberExtension objects are used by components that display the contents of CyberItem objects; this includes browser-like components, such as a Gopher browser or Web browser, along with viewer-like components, such as JPEG, MPEG or text viewers. The CyberExtension objects also keep track of the CyberItem objects which these components are responsible for displaying. In accordance with the invention, the class GopherBrowser **1010** may be used to construct a Gopher-like network browsing component and the class WebBrowser **1012** may be used to construct a Web-like network browsing component.

FIGS. 11A–11D are highly schematized diagrams illustrating the interactions between the novel network-oriented components, including the encapsulated (CyberItem) network entity component according to the invention. It is to be understood that the components described herein are objects constructed from the interconnected abstract classes. In general, a user has "double clicked" on an icon of a graphical user interface **1100** displayed on a computer screen. The icon represents, e.g., a Gopher directory displayed in a Gopher browser application. Initially, a GopherBrowser component **1110** displays two icons representing CyberItem components, the icons labeled (GopherItem) Happy Face **1104** and (GopherItem) Home Page **1106**. These latter components represent the contents of a Gopher directory labeled (GopherItem) Apple **1114**.

In FIG. 11A, the left side of the diagram illustrates a GopherBrowser component **1110** that is displayed on the computer screen, i.e., the right side of the diagram. The GopherBrowser component has a CyberExtension component **1112** which keeps track of the GopherItem components. When the user double clicks on the Home Page GopherItem icon **1106**, the GopherBrowser component **1110** receives this event and issues a call to an "Open" method of a Home Page GopherItem component; this call instructs the GopherItem component **1106** to open itself.

Specifically, and referring to FIG. 11B, the GopherItem component **1106** creates a component of the appropriate type to display itself. For this example, the GopherItem preferably creates a WebBrowser component **1120**. Once created, the WebBrowser component further creates a CyberExtension component **1122** for storing the Home Page GopherItem component (now shown at **1124**). In accordance with the invention, the Home Page GopherItem component is a network entity containing a pointer that points to the network address of a Gopher server storing the appropriate Web page.

In FIG. 11C, the CyberExtension component 1122 then notifies the WebBrowser component 1120 that it has been assigned a GopherItem component 1124 to display. The WebBrowser component 1120 calls a method CreateCyberStream of the GopherItem to create a GopherStream component 1130 for downloading the appropriate data. Thereafter, the WebBrowser component 1120 begins asychronously downloading an HTML document from the appropriate Gopher server (not shown).

Control of the execution of this process then returns to the GopherItem component 1124 in FIG. 11D. This component, in turn, issues a call to an Open method of the WebBrowser component 1120, which causes the downloaded HTML document to appear on the screen (now shown at 1140). For a further understanding of the invention, FIG. 12 provides an illustrative flowchart 1200 of the sequence of steps involved in invoking, and accessing, information from a referenced network resource, as described above.

In summary, the network-oriented component system provides a customizable framework that enables a user to create an encapsulated entity containing a reference to a network resource on a computer network. Advantageously, the inventive encapsulation technique allows a user to simply manipulate visual objects when accessing information on the network. Instead of having to type the destination address of a resource, the user can merely "drag and drop" the icon associated with entity anywhere on the graphical user interface. When the user "double clicks" on the icon, the entity opens up in a window and displays the contents of the resource at that network location. Since the address is encapsulated within the network reference entity, the user does not have to labor with typing of the cumbersome character string.

While there has been shown and described an illustrative embodiment for implementing an extensible and replaceable network component system, it is to be understood that various other adaptations and modifications may be made within the spirit and scope of the invention. For example, additional system software routines may be used when implementing the invention in various applications. These additional system routines include dynamic link libraries (DLL), which are program files containing collections of window environment and networking functions designed to perform specific classes of operations. These functions are invoked as needed by the software component layer to perform the desired operations. Specifically, DLLs, which are generally well-known, may be used to interact with the component layer and window manager to provide network-specific components and functions.

The foregoing description has been directed to specific embodiments of this invention. It will be apparent, however, that other variations and modifications may be made to the described embodiments, with the attainment of some or all of their advantages. Therefore, it is the object of the appended claims to cover all such variations and modifications as come within the true spirit and scope of the invention.

What is claimed is:

1. A method of efficiently accessing information from a network resource located on a computer network for display on a computer coupled to the network, the network resource having one or more associated data types, each data type being accessible by a corresponding object-oriented software component, the method comprising the steps of:

defining at least one network component that integrates the object-oriented software components needed to access the one or more data types associated with the network resource;

creating an encapsulated entity component containing a reference to a location of the network resource on the computer network, the encapsulated entity component also identifying the at least one network component that was defined for the network resource;

storing the encapsulated entity component as a visual object on the computer;

in response to manipulation of the visual object with a pointing device, displaying the contents of the network resource on a screen of the computer by invoking the object-oriented software components integrated by the at least one identified network component.

2. The method of claim 1 wherein the step of displaying comprises the step of invoking a first network component for displaying the contents of the referenced network resource on the screen, the first network component comprising a browsing component.

3. The method of claim 2 wherein the step of displaying further comprises the step of invoking a second network component for transfering the contents of the referenced network resource to the first network component, the second network component comprising a data stream component.

4. The method of claim 3 further comprising the step of creating objects for communication among the encapsulated entity and network components through application programming interfaces.

5. The method of claim 4 wherein the step of creating comprises the step of constructing the encapsulated entity component from an Item object defined by an Item object class.

6. The method of claim 5 wherein the step of creating comprises the step of spawning a Stream object from the Item object, the Stream object representing the data stream.

7. Apparatus for efficiently accessing information from a network resource located on a computer network for display on a computer coupled to the network, the network resource having one or more associated data types, each data type being accessible by a corresponding object-oriented software component, the apparatus comprising:

an object-oriented software component architecture layer configured to define at least one network component that integrates the object-oriented software components needed to access the one or more data types associated with the network resource; and

an encapsulated network entity component cooperating with the component architecture layer and containing a reference to the network resource and an identifier for the at least one network component that was defined for the network resource wherein, the encapsulated network entity component is manifested as visual object on a display screen of the computer and further wherein, the encapsulated network entity component is adapted for manipulation by a pointing device of the computer to display contents of the network resource on the screen by invoking the object-oriented software components integrated by the at least one identified network component.

8. The apparatus of claim 7 further comprising:

an operating system interfacing with the component architecture layer to control the operations of the computer; and

a network component layer coupled to the component architecture layer to form a cooperating component computing arrangement.

9. The apparatus of claim 8 wherein the cooperating component computing arrangement generates the encapsulated network entity.

10. The apparatus of claim 9 wherein the reference to the network resource is a pointer that identifies the address of the network resource on a computer network.

11. The apparatus of claim 10 wherein the pointer is a uniform resource locator.

12. The apparatus of claim 11 wherein the uniform resource locator has a first portion that identifies the network resource and a second portion that specifies a means for accessing that resource.

13. The apparatus of claim 11 wherein the uniform resource locator is a character string that describes a protocol used to address the network resource, a server on which the resource resides, a path to the resource and a resource filename.

14. The apparatus of claim 10 wherein the pointer is a post office protocol account.

15. Apparatus for efficiently accessing information from a network resource located on a computer network for display on a computer coupled to the network, the network resource having one or more associated data types, each data type being accessible by a corresponding object-oriented software component, the apparatus comprising:

means for defining at least one network component that integrates the object-oriented software components needed to access the one or more data types associated with the network resource;

means for creating an encapsulated entity component containing a reference to a location of the network resource on the computer network, the encapsulated entity component also identifying the at least one network component that was defined for the network resource;

means for storing the encapsulated entity component as a visual object on the computer; and

means, responsive to manipulation of the visual object with a pointing device, for displaying contents of the network resource on a screen of the computer by invoking the object-oriented software components integrated by the at least one identified network component.

16. The apparatus of claim 15 wherein the means for displaying comprises means for invoking a first network component for displaying the contents of the referenced network resource on the screen, the first network component comprising a browsing component.

17. The apparatus of claim 16 wherein the means for displaying further comprises means for invoking a second network component for transfering the contents of the referenced network resource to the first network component, the second network component comprising a data stream component.

18. The apparatus of claim 17 further comprising means for creating objects for communication among the encapsulated entity and network components through application programming interfaces.

19. The apparatus of claim 18 wherein the means for creating comprises means for constructing the encapsulated entity component from an Item objected defined by an Item object class.

20. The apparatus of claim 19 wherein the means for creating comprises means for spawning a Stream object from the Item object, the Stream object representing the data stream.

* * * * *

EXHIBIT J

U 1670074

THE UNITED STATES OF AMERICA

TO ALL TO WHOM THESE PRESENTS SHALL COME:

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office

November 16, 2007

THIS IS TO CERTIFY THAT ANNEXED HERETO IS A TRUE COPY FROM THE RECORDS OF THIS OFFICE OF:

U.S. PATENT: *5,915,131*
ISSUE DATE: *June 22, 1999*

By Authority of the
Under Secretary of Commerce for Intellectual Property
and Director of the United States Patent and Trademark Office

P. SWAIN
Certifying Officer

UNITED STATES PATENT AND TRADEMARK OFFICE
DEPARTMENT OF COMMERCE

US005915131A

United States Patent [19]

Knight et al.

[11] Patent Number: 5,915,131

[45] Date of Patent: Jun. 22, 1999

[54] **METHOD AND APPARATUS FOR HANDLING I/O REQUESTS UTILIZING SEPARATE PROGRAMMING INTERFACES TO ACCESS SEPARATE I/O SERVICES**

[75] Inventors: **Holly N. Knight**, La Honda; **Carl D. Sutton**, Palo Alto; **Wayne N. Meretsky**, Los Altos; **Alan B. Mimms**, San Jose, all of Calif.

[73] Assignee: **Apple Computer, Inc.**, Cupertino, Calif.

[21] Appl. No.: **08/435,677**

[22] Filed: **May 5, 1995**

[51] **Int. Cl.**^{6} **G06F 9/40**; G06F 13/14

[52] **U.S. Cl.** **395/892**; 395/682; 395/828; 395/702; 707/104; 345/333

[58] **Field of Search** 395/828, 702, 395/834, 200.2, 892, 682, 309; 345/333; 707/104

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,593,352	6/1986	Castel et al.	364/200
4,727,537	2/1988	Nichols	370/85
4,908,859	3/1990	Bennett et al.	380/10
4,982,325	1/1991	Tignor et al.	364/200
5,129,086	7/1992	Coyle, Jr. et al.	395/650
5,148,527	9/1992	Basso et al.	395/325
5,197,143	3/1993	Lary et al.	395/425
5,430,845	7/1995	Rimmer et al.	395/275
5,491,813	2/1996	Bondy et al.	395/500
5,513,365	4/1996	Cook et al.	395/800
5,535,416	7/1996	Feeney et al.	395/834
5,537,466	7/1996	Taylor et al.	379/201
5,553,245	9/1996	Su et al.	395/284
5,572,675	11/1996	Bergler	395/200.2

OTHER PUBLICATIONS

Forin, A., et al. entitled "An I/O System for Mach 3.0," Proceedings of the Usenix Mach Symposium 20–22, Nov. 1991, Monterey, CA, US, 20–22 Nov. 1991, pp. 163–176.

Steve Lemon and Kennan Rossi, entitled "An Object Oriented Device Driver Model," Digest of Papers Compcon '95, Technologies for the Information Superhighway 5–9, Mar. 1995, San Francisco, CA, USA pp. 360–366.

Glenn Andert, entitled "Object Frameworks in the Taligent OS," Intellectual Leverage: Digest of Papers of the Spring Computer SOCI International Conference (Compcon), San Francisco, Feb. 28–Mar. 4, 1994, Feb. 24, 1994, Institute of Electrical and Electronics Engineers, pp. 112–121.

Hu, 'Interconnecting electronic mail networks: Gateways and translation strategies are proposed for backbone networks to interchange incompatible electronic documents on multivendor networks', Data Communications, p. 128, vol. 17, No. 10, Sep. 1988.

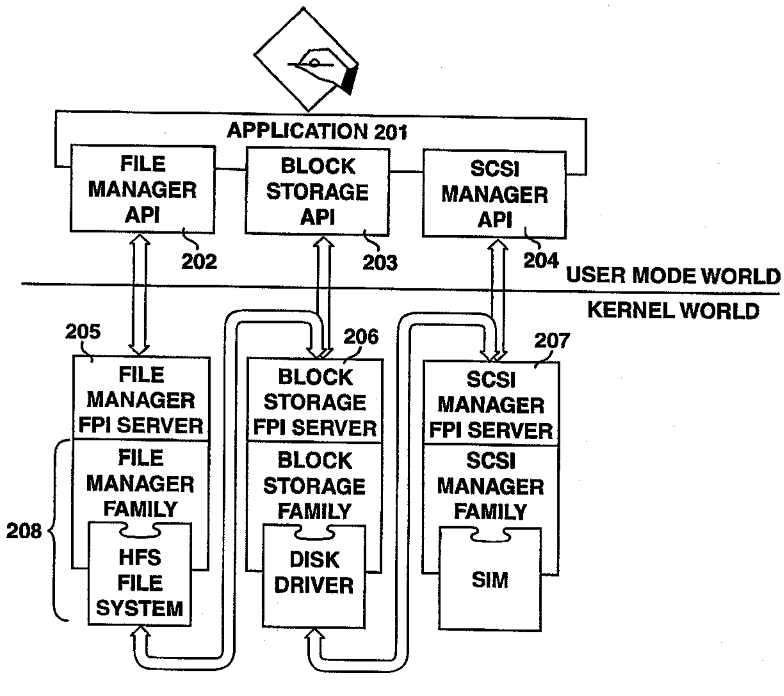
Knibbe, 'IETF's Resource Reservation Protocol to facilitate mixed voice, data, and video nets', Network World, p. 51, Apr. 24, 1995.

Primary Examiner—Thomas C. Lee
Assistant Examiner—Rehana Perveen
Attorney, Agent, or Firm—Blakely, Sokoloff, Taylor & Zafman

[57] **ABSTRACT**

A computer system handling multiple applications wherein groups of I/O services are accessible through separate application programming interfaces. Each application has multiple application programming interfaces by which to access different families of I/O services, such as I/O devices.

20 Claims, 8 Drawing Sheets



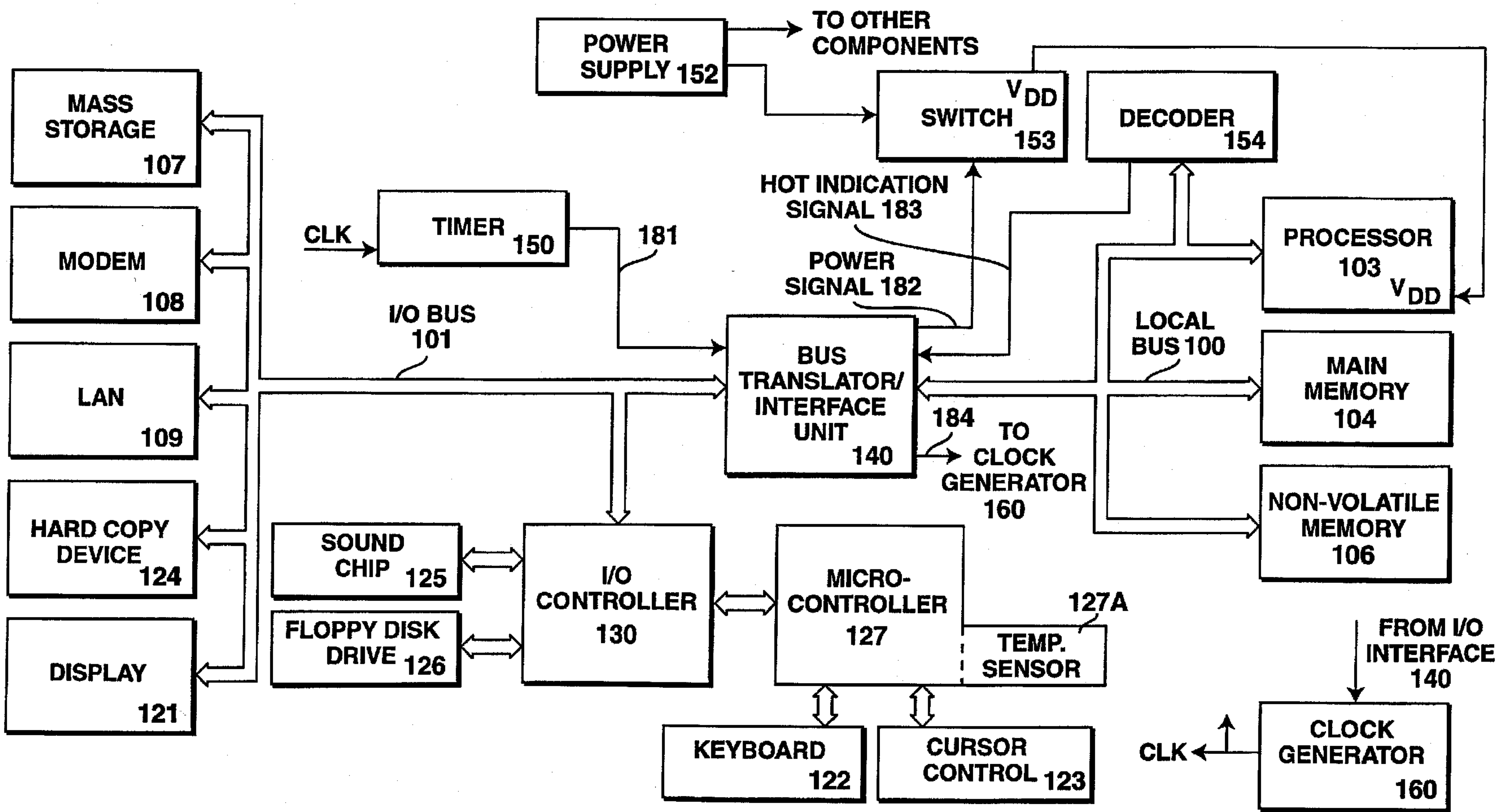

POWER SUPPLY 152
TO OTHER COMPONENTS
SWITCH 153
VDD
DECODER 154
MASS STORAGE 107
MODEM 108
LAN 109
HARD COPY DEVICE 124
DISPLAY 121
CLK
TIMER 150
181
HOT INDICATION SIGNAL 183
POWER SIGNAL 182
PROCESSOR 103
VDD
I/O BUS 101
LOCAL BUS 100
BUS TRANSLATOR/ INTERFACE UNIT 140
MAIN MEMORY 104
184
TO CLOCK GENERATOR 160
NON-VOLATILE MEMORY 106
SOUND CHIP 125
FLOPPY DISK DRIVE 126
I/O CONTROLLER 130
MICRO-CONTROLLER 127
127A
TEMP. SENSOR
FROM I/O INTERFACE 140
KEYBOARD 122
CURSOR CONTROL 123
CLK
CLOCK GENERATOR 160

FIG. 1

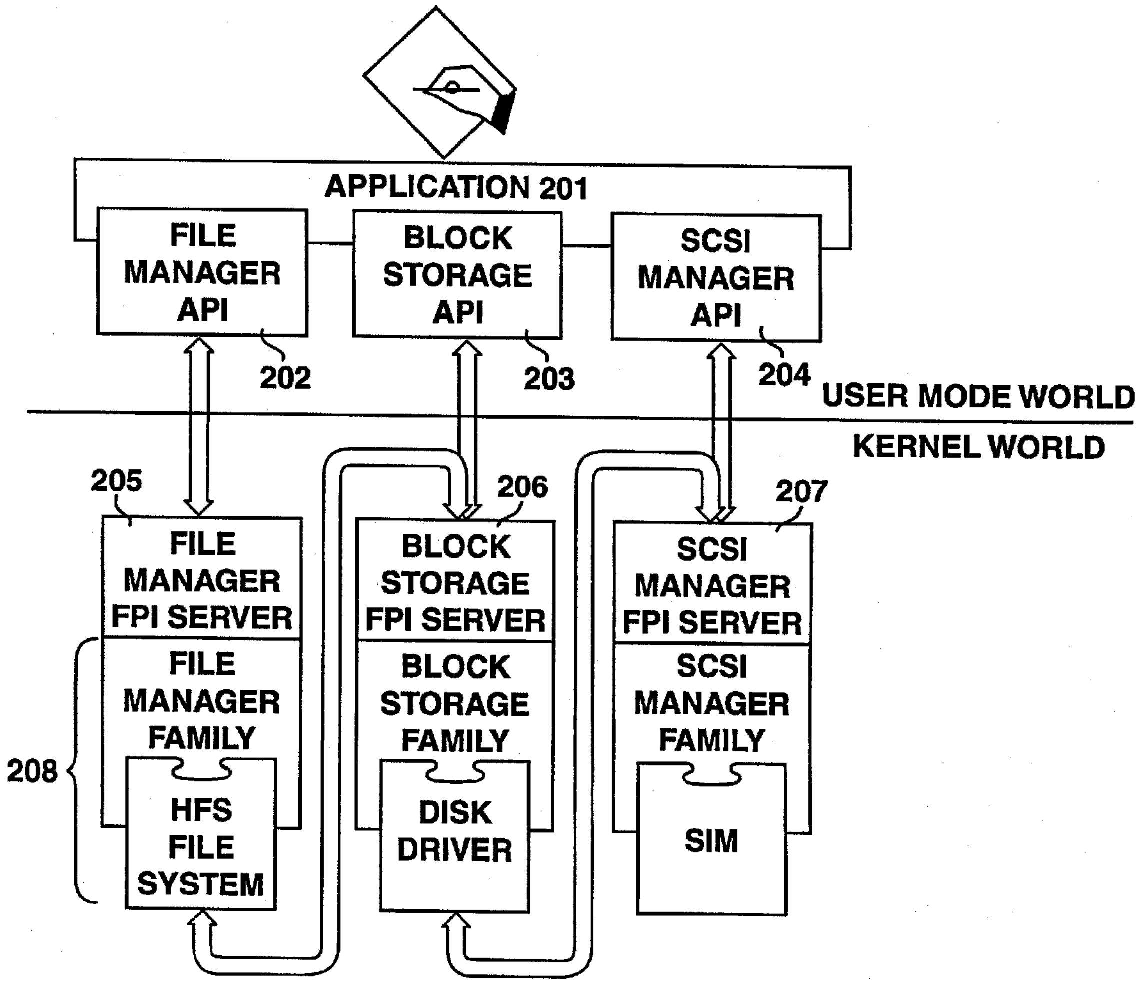

APPLICATION 201
FILE MANAGER API
BLOCK STORAGE API
SCSI MANAGER API
202
203
204
USER MODE WORLD
KERNEL WORLD
205
206
207
FILE MANAGER FPI SERVER
BLOCK STORAGE FPI SERVER
SCSI MANAGER FPI SERVER
FILE MANAGER FAMILY
BLOCK STORAGE FAMILY
SCSI MANAGER FAMILY
208
HFS FILE SYSTEM
DISK DRIVER
SIM

FIG. 2

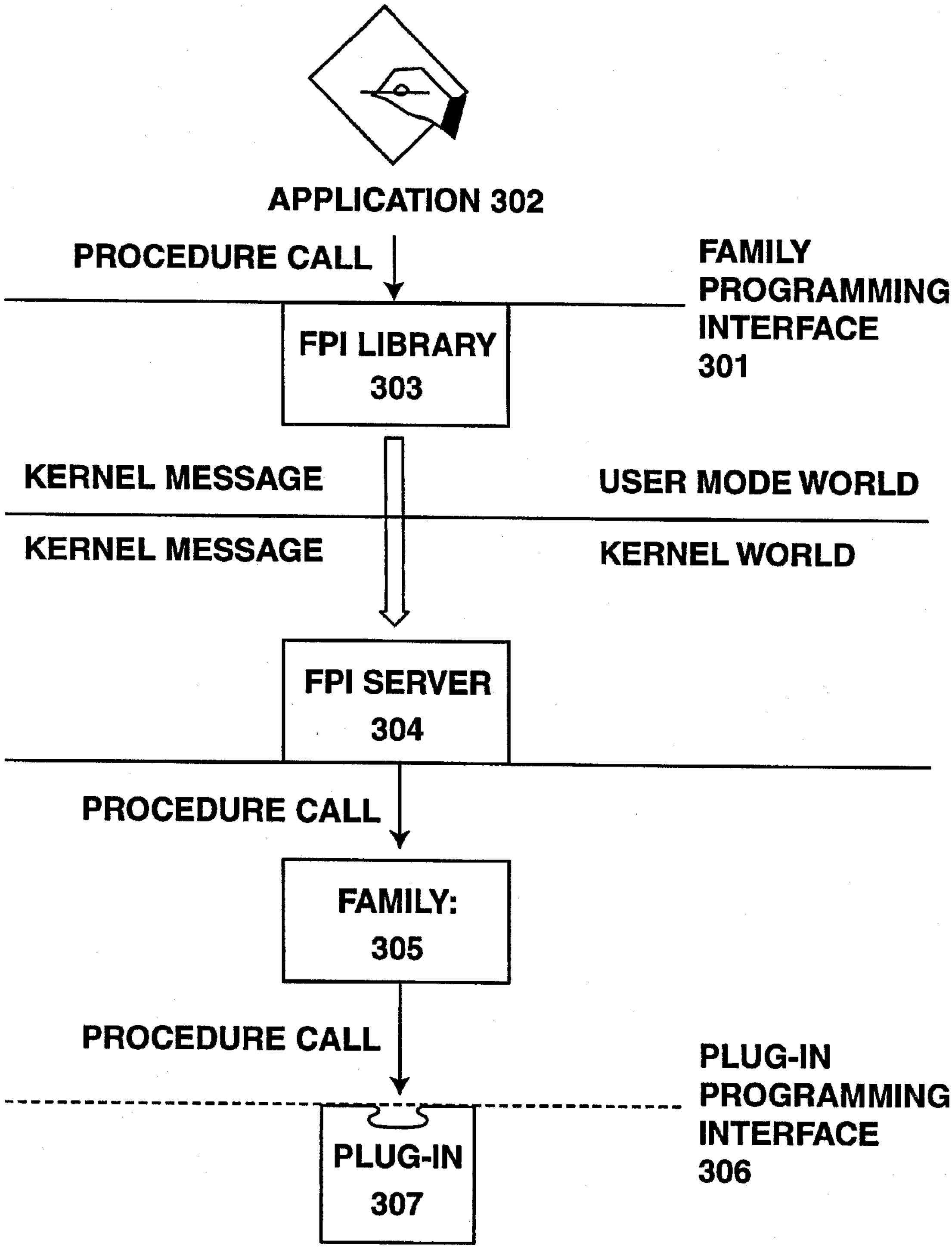
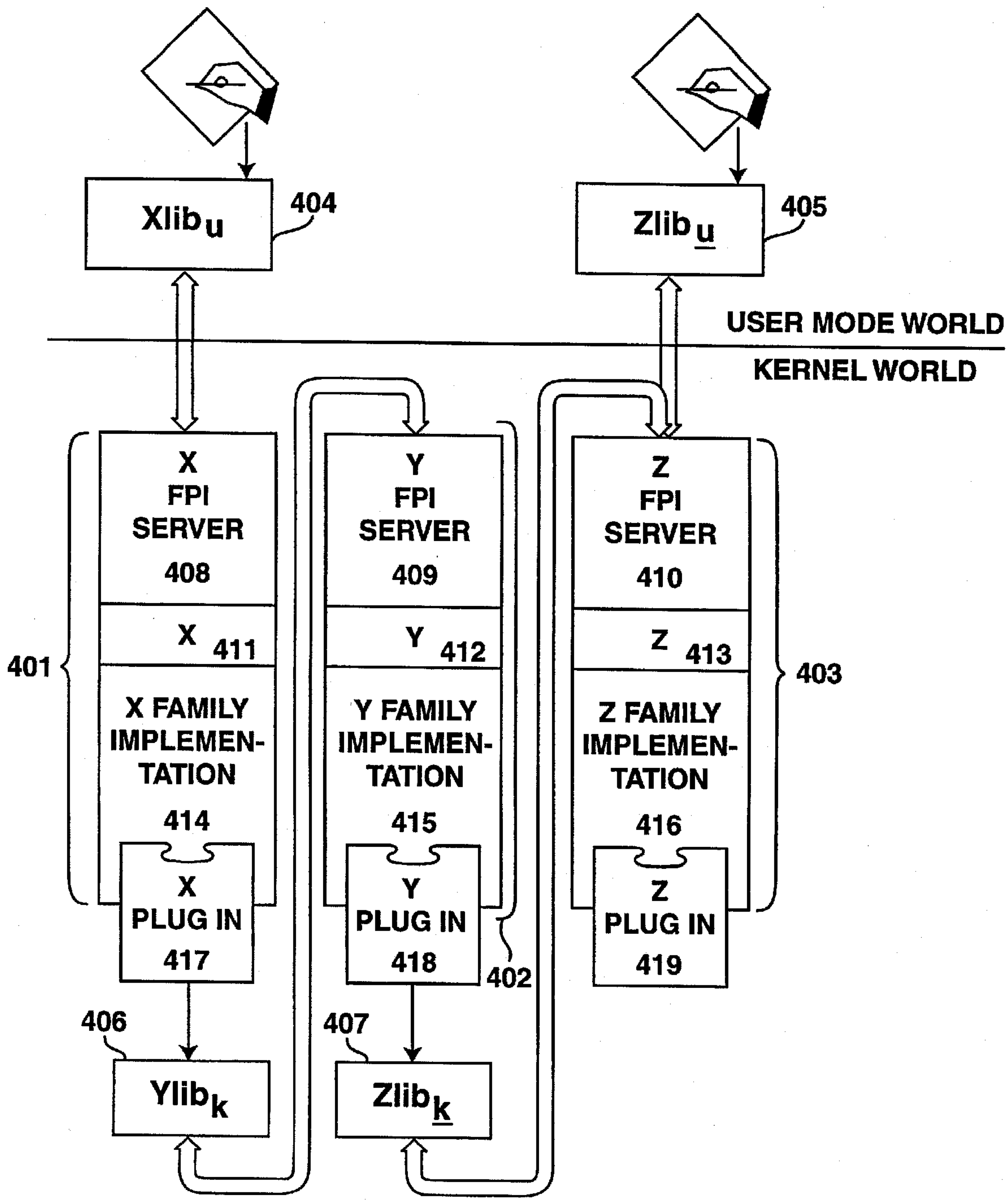
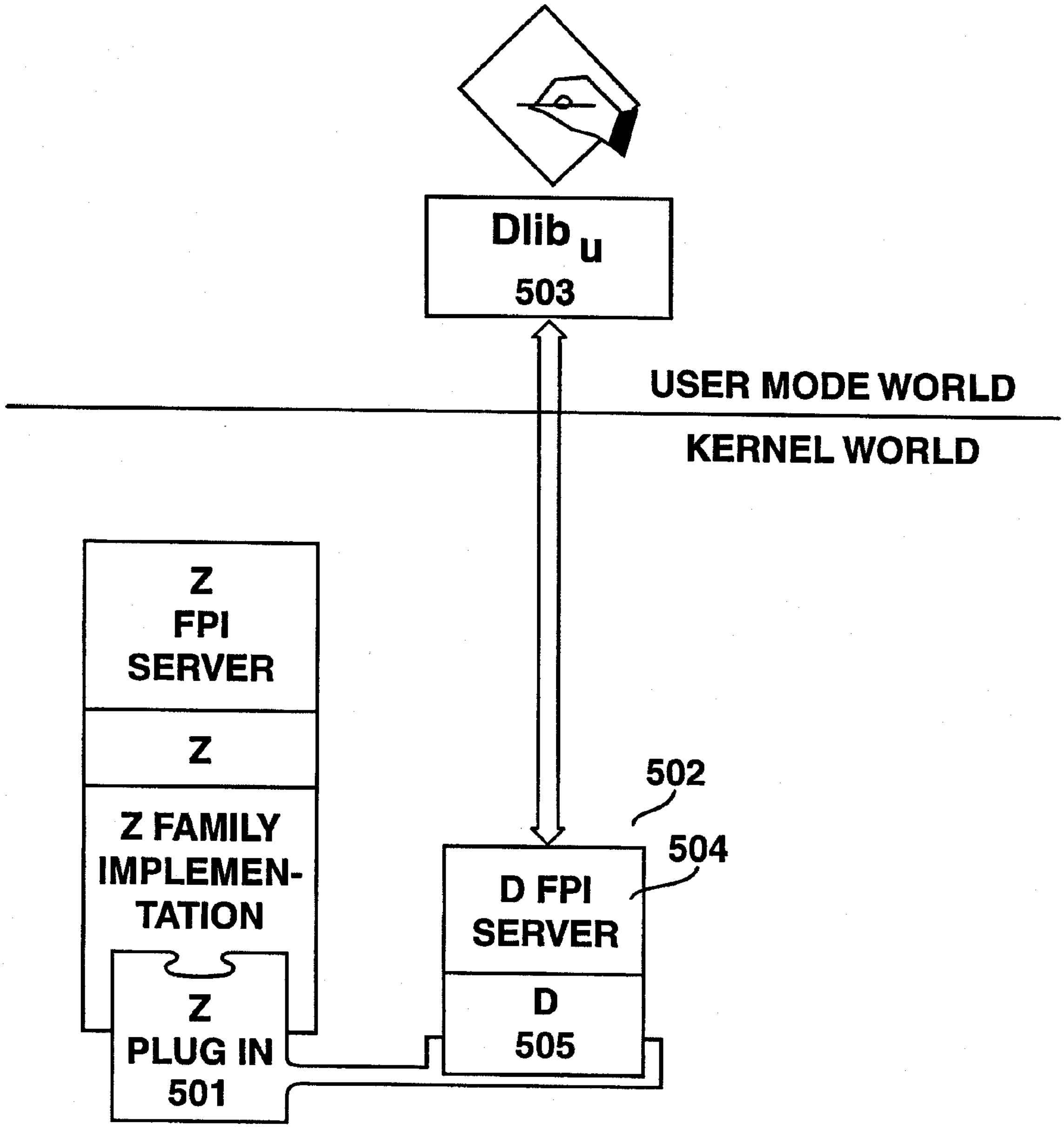

APPLICATION 302
PROCEDURE CALL
FAMILY PROGRAMMING INTERFACE 301
FPI LIBRARY 303
KERNEL MESSAGE
USER MODE WORLD
KERNEL MESSAGE
KERNEL WORLD
FPI SERVER 304
PROCEDURE CALL
FAMILY: 305
PROCEDURE CALL
PLUG-IN PROGRAMMING INTERFACE 306
PLUG-IN 307

FIG. 3



Xlib u
404
Zlib u
405
USER MODE WORLD
KERNEL WORLD
X
FPI
SERVER
408
Y
FPI
SERVER
409
Z
FPI
SERVER
410
X 411
Y 412
Z 413
401
403
X FAMILY
IMPLEMEN-
TATION
414
Y FAMILY
IMPLEMEN-
TATION
415
Z FAMILY
IMPLEMEN-
TATION
416
X
PLUG IN
417
Y
PLUG IN
418
402
Z
PLUG IN
419
406
407
Ylib k
Zlib k

FIG. 4



Dlib u
503
USER MODE WORLD
KERNEL WORLD
Z
FPI
SERVER
Z
Z FAMILY
IMPLEMEN-
TATION
Z
PLUG IN
501
502
504
D FPI
SERVER
D
505

FIG. 5

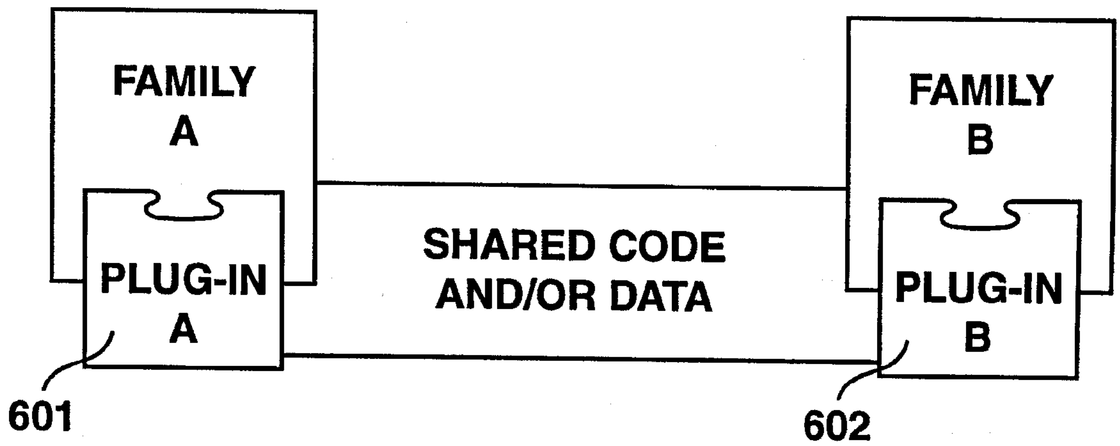

FAMILY
A
PLUG-IN
A
601
SHARED CODE
AND/OR DATA
FAMILY
B
PLUG-IN
B
602

FIG. 6

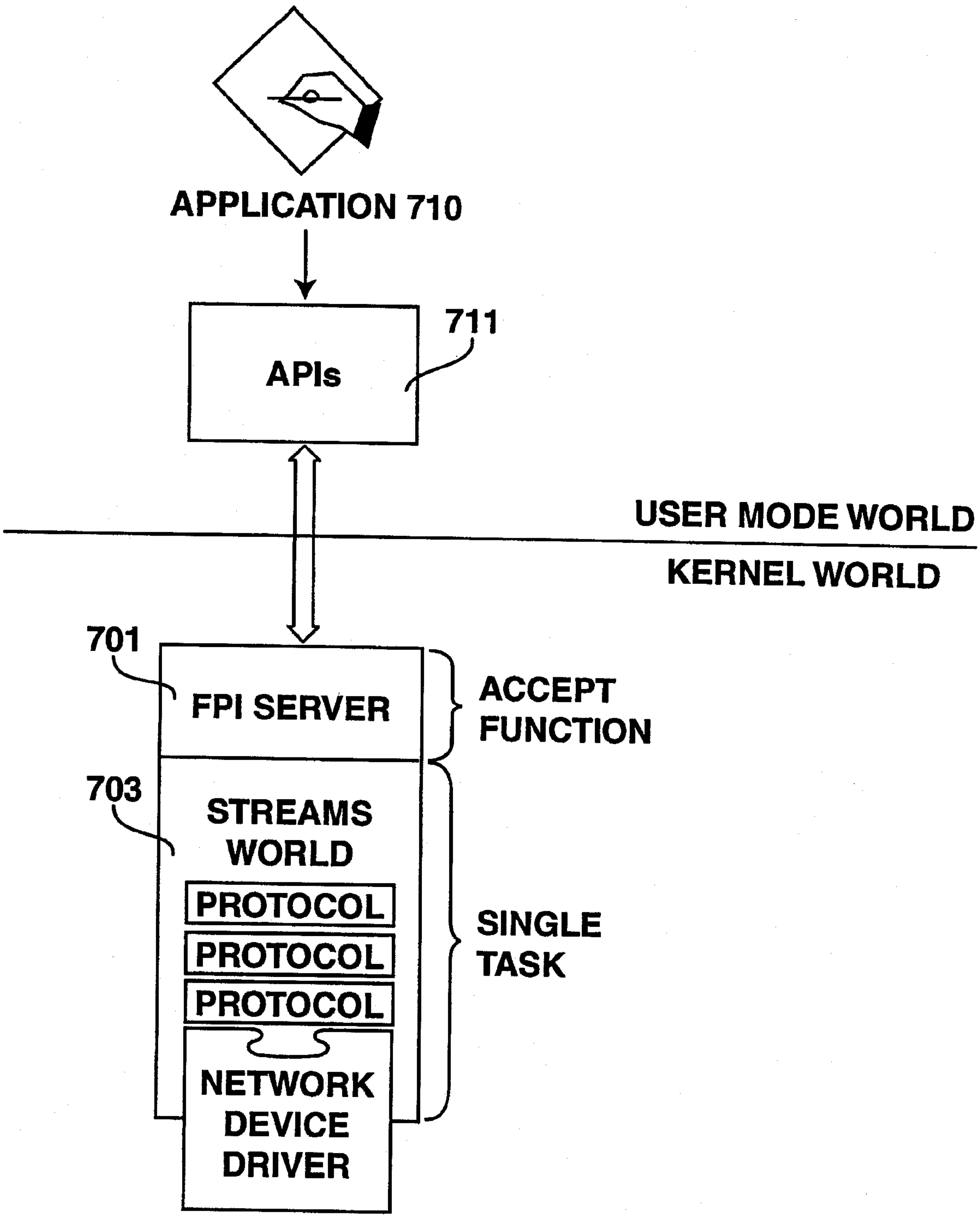
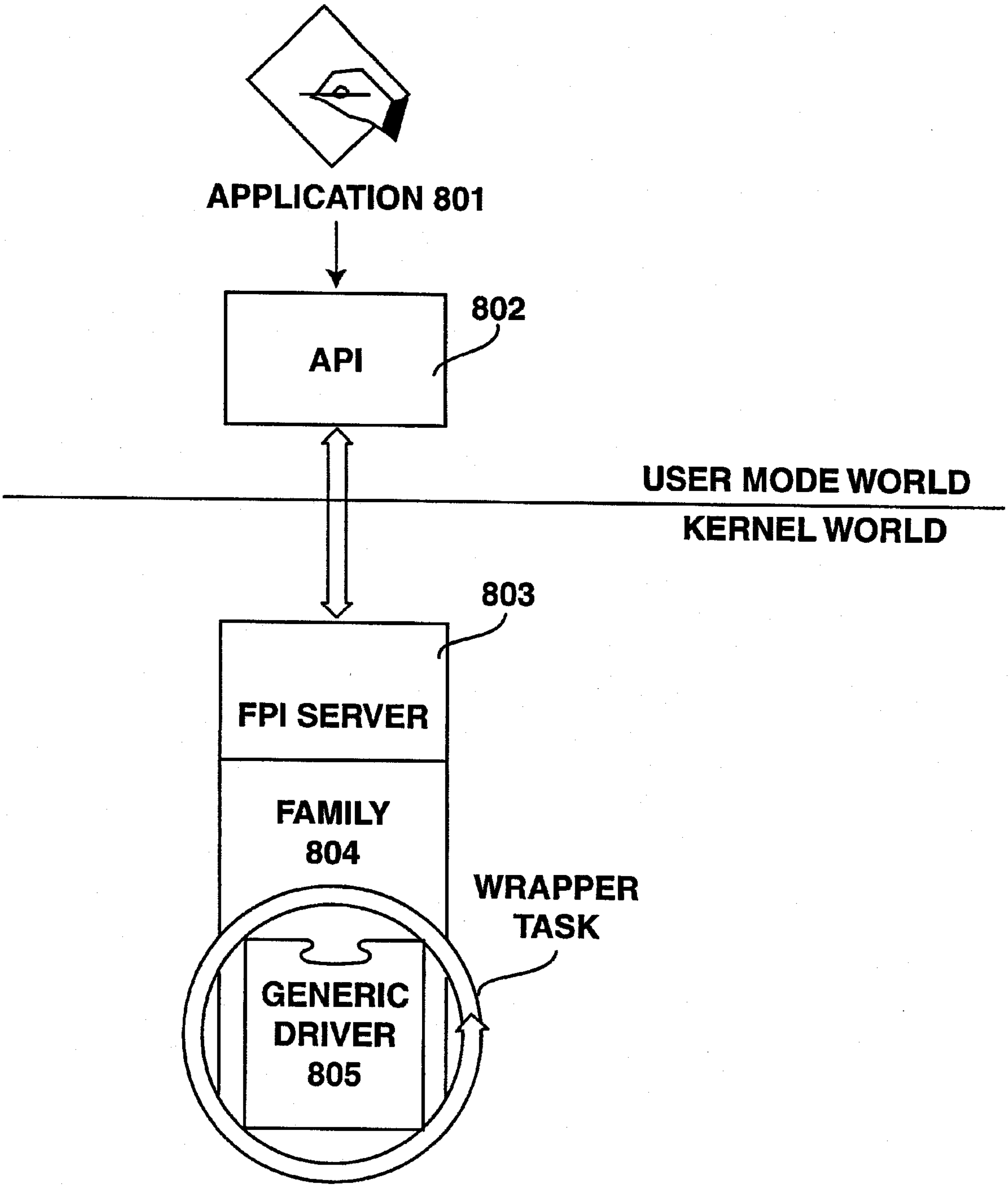

APPLICATION 710
APIs
711
USER MODE WORLD
KERNEL WORLD
701
FPI SERVER
ACCEPT FUNCTION
703
STREAMS WORLD
PROTOCOL
PROTOCOL
PROTOCOL
NETWORK DEVICE DRIVER
SINGLE TASK

FIG. 7



APPLICATION 801
API
802
USER MODE WORLD
KERNEL WORLD
803
FPI SERVER
FAMILY
804
WRAPPER
TASK
GENERIC
DRIVER
805

FIG. 8

METHOD AND APPARATUS FOR HANDLING I/O REQUESTS UTILIZING SEPARATE PROGRAMMING INTERFACES TO ACCESS SEPARATE I/O SERVICES

FIELD OF THE INVENTION

The invention relates to the field of computer systems; particularly, the present invention relates to handling service requests generated by application programs.

BACKGROUND OF THE INVENTION

Application programs running in computer systems often access system resources, such as input/output (I/O) devices. These system resources are often referred to as services. Certain sets of services (e.g., devices) have similar characteristics. For instance, all display devices or all ADB devices have similar interface requirements.

To gain access to I/O resources, applications generate service requests to which are sent through an application programming interface (API). The service requests are converted by the API to a common set of functions that are forwarded to the operating system to be serviced. The operating system then sees that service requests are responded to by the appropriate resources (e.g., device). For instance, the operating system may direct a request to a device driver.

One problem in the prior art is that service requests are not sent directly to the I/O device or resource. All service requests from all applications are typically sent through the same API. Because of this, all of the requests are converted into a common set of functions. These common set of functions do not have meaning for all the various types of I/O devices. For instance, a high level request to play a sound may be converted into a write function to a sound device. However, the write function is not the best method of communicating sound data to the sound device. Thus, another conversion of write data to a sound data format may be required. Also, some functions do not have a one-to-one correspondence with the function set of some I/O devices. Thus, it would be desirable to avoid this added complexity and to take advantage of the similar characteristics of classes of I/O devices when handling I/O requests, while providing services and an environment in which to run those services that is tuned to the specific device needs and requirements.

SUMMARY OF THE INVENTION

A method and apparatus for handling I/O requests is described. In the present invention, the I/O requests are handled by the computer system having a bus and a memory coupled to the bus that stores data and programming instructions. The programming instructions include application programs and an operating system. A processing unit is coupled to the bus and runs the operating system and application programs by executing programming instructions. Each application programs have multiple separate programming interfaces available to access multiple sets of I/O services provided through the operating system via service requests.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention will be understood more fully from the detailed description given below and from the accompanying drawings of various embodiments of the invention, which, however, should not be taken to limit the invention to the specific embodiments, but are for explanation and understanding only.

FIG. 1 a block diagram of one embodiment in the computer system of the present invention.

FIG. 2 is an overview of the I/O architecture of the present invention.

FIG. 3 illustrates a flow diagram of I/O service request handling according to the teachings of the present invention.

FIG. 4 illustrates an overview of the I/O architecture of the present invention having selected families accessing other families.

FIG. 5 illustrates extended programming family interface of the present invention.

FIG. 6 illustrates plug-in modules of different families that share code and/or data.

FIG. 7 illustrates a single task activation model according to the teachings of the present invention.

FIG. 8 illustrates a task-per-plug-in model used as an activation model according to the teachings of the present invention.

DETAILED DESCRIPTION OF THE PRESENT INVENTION

A method and apparatus handling service requests is described. In the following detailed description of the present invention numerous specific details are set forth in order to provide a thorough understanding of the present invention. However, it will be apparent to one skilled in the art that the present invention may be practiced without these specific details. In other instances, well-known structures and devices are shown in block diagram form, rather than in detail, in order to avoid obscuring the present invention.

Some portions of the detailed descriptions which follow are presented in terms of algorithms and symbolic representations of operations on data bits within a computer memory. These algorithmic descriptions and representations are the means used by those skilled in the data processing arts to most effectively convey the substance of their work to others skilled in the art. An algorithm is here, and generally, conceived to be a self-consistent sequence of steps leading to a desired result. The steps are those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated. It has proven convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like.

It should be borne in mind, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities. Unless specifically stated otherwise as apparent from the following discussions, it is appreciated that throughout the present invention, discussions utilizing terms such as "processing" or "computing" or "calculating" or "determining" or "displaying" or the like, refer to the action and processes of a computer system, or similar electronic computing device, that manipulates and transforms data represented as physical (electronic) quantities within the computer system's registers and memories into other data similarly represented as physical quantities within the computer system memories or registers or other such information storage, transmission or display devices.

The present invention also relates to apparatus for performing the operations herein. This apparatus may be specially constructed for the required purposes, or it may

comprise a general purpose computer selectively activated or reconfigured by a computer program stored in the computer. The algorithms and displays presented herein are not inherently related to any particular computer or other apparatus. Various general purpose machines may be used with programs in accordance with the teachings herein, or it may prove convenient to construct more specialized apparatus to perform the required method steps. The required structure for a variety of these machines will appear from the description below. In addition, the present invention is not described with reference to any particular programming language. It will be appreciated that a variety of programming languages may be used to implement the teachings of the invention as described herein.

Overview of the Computer System of the Present Invention

Referring to FIG. 1, an overview of a computer system of the present invention is shown in block diagram form. The present invention may be implemented on a general purpose microcomputer, such as one of the members of the Apple family of personal computers, one of the members of the IBM personal computer family, or one of several other computer devices which are presently commercially available. Of course, the present invention may also be implemented on a multi-user system while encountering all of the costs, speed, and function advantages and disadvantages available with these machines.

As illustrated in FIG. 1, the computer system of the present invention generally comprises a local bus or other communication means **100** for communicating information, a processor **103** coupled with local bus **100** for processing information, a random access memory (RAM) or other dynamic storage device **104** (commonly referred to as a main memory) coupled with local bus **100** for storing information and instructions for processor **103**, and a read-only memory (ROM) or other non-volatile storage device **106** coupled with local bus **100** for storing non-volatile information and instructions for processor **103**.

The computer system of the present invention also includes an input/output (I/O) bus or other communication means **101** for communication information in the computer system. A data storage device **107**, such as a magnetic tape and disk drive, including its associated controller circuitry, is coupled to I/O bus **101** for storing information and instructions. A display device **121**, such as a cathode ray tube, liquid crystal display, etc., including its associated controller circuitry, is also coupled to I/O bus **101** for displaying information to the computer user, as well as a hard copy device **124**, such as a plotter or printer, including its associated controller circuitry for providing a visual representation of the computer images. Hard copy device **124** is coupled with processor **103**, main memory **104**, non-volatile memory **106** and mass storage device **107** through I/O bus **101** and bus translator/interface unit **140**. A modem **108** and an ethernet local area network **109** are also coupled to I/O bus **101**.

Bus interface unit **140** is coupled to local bus **100** and I/O bus **101** and acts as a gateway between processor **103** and the I/O subsystem. Bus interface unit **140** may also provide translation between signals being sent from units on one of the buses to units on the other bus to allow local bus **100** and I/O bus **101** to co-operate as a single bus.

An I/O controller **130** is coupled to I/O bus **101** and controls access to certain I/O peripherals in the computer system. For instance, I/O controller **130** is coupled to controller device **127** that controls access to an alpha-numeric input device **122** including alpha-numeric and other keys, etc., for communicating information and command

selections to processor **103**, and a cursor control **123**, such as a trackball, stylus, mouse, or trackpad, etc., for controlling cursor movement. The system also includes a sound chip **125** coupled to I/O controller **130** for providing audio recording and play back. Sound chip **125** may include a sound circuit and its driver which are used to generate various audio signals from the computer system. I/O controller **130** may also provide access to a floppy disk and driver **126**. The processor **103** controls I/O controller **130** with its peripherals by sending commands to I/O controller **130** via local bus **100**, interface unit **140** and I/O bus **101**.

Batteries or other power supply **152** may also be included to provide power necessary to run the various peripherals and integrated circuits in the computer system. Power supply **152** is typically a DC power source that provides a constant DC power to various units, particularly processor **103**. Various units such as processor **103**, display **121**, etc., also receive clocking signals to synchronize operations within the computer systems. These clocking signals may be provided by a global clock generator or multiple clock generators, each dedicated to a portion of the computer system. Such a clock generator is shown as clock generator **160**. In one embodiment, clock generator **160** comprise a phase-locked loop (PLL) that provides clocking signals to processor **103**.

I/O controller **140** includes control logic to coordinate the thermal management. Several additional devices are included within the computer system to operate with the control logic within I/O controller **140**. A timer **150**, a switch **153** and a decoder **154** are included to function in connection with the control logic. In one embodiment, decoder **154** is included within bus interface unit **140** and timer **150** is included in I/O controller **130**.

Switch **153** is a p-channel power MOSFET, which has its gate connected to the power signal **182**, its source to the power supply and its drain to processor's V_{DD} pin.

In one embodiment, processor **103** is a member of the PowerPC™ family of processors, such as those manufactured by Motorola Corporation of Schaumberg, Ill. The memory in the computer system is initialized to store the operating system as well as other programs, such as file directory routines and application programs, and data inputted from I/O controller **130**. In one embodiment, the operating system is stored in ROM **106**, while RAM **104** is utilized as the internal memory for the computer system for accessing data and application programs. Processor **103** accesses memory in the computer system via an address bus within bus **100**. Commands in connection with the operation of memory in the computer system are also sent from the processor to the memory using bus **100**. Bus **100** also includes a bi-directional data bus to communicate data in response to the commands provided by processor **103** under the control of the operating system running on it.

Of course, certain implementations and uses of the present invention may neither require nor include all of the above components. For example, in certain implementations a keyboard or cursor control device for inputting information to the system may not be required. In other implementations, it may not be required to provide a display device displaying information. Furthermore, the computer system may include additional processing units.

The operating system running on processor **103** takes care of basic tasks such as starting the system, handling interrupts, moving data to and from memory **104** and peripheral devices via input/output interface unit **140**, and managing the memory space in memory **104**. In order to take care of such operations, the operating system provides

multiple execution environments at different levels (e.g., task level, interrupt level, etc.). Tasks and execution environments are known in the art.

Overview of the Present Invention

In one embodiment, the computer system runs a kernel-based, preemptive, multitasking operation system in which applications and I/O services, such as drivers, operate in separate protection domains (e.g., the user and kernel domains, respectively). The user domain does not have direct access to data of the kernel domain, while the kernel domain can access data in the user domain.

The computer system of the present invention uses one or more separate families to provide I/O services to the system. Each I/O family provides a set of I/O services to the system. For instance, a SCSI family and its SCSI interface modules (SIMs) provide SCSI based services, while a file systems family and its installable file systems provide file management services. In one embodiment, an I/O family is implemented by multiple modules and software routines.

Each family defines a family programming interface (FPI) designed to meet the particular needs of that family. An FPI provides access to a given family's plug-ins, which are dynamically loaded pieces of software that each provide an instance of the service provided by a family. For example, within the file systems family (File Manager), a plug-in implements file-system-specific services. In one embodiment, plug-ins are a superset of device drivers, such that all drivers are plug-ins, but not all plug-ins are drivers.

Access to services is available only through an I/O family's programming interface. In one embodiment, hardware is not directly accessible to application software, nor is it vulnerable to application error. Applications have access to hardware services only through an I/O family's programming interface. Also, the context within which an I/O service runs and the method by which it interacts with the system is defined by the I/O family to which it belongs.

FIG. 2 illustrates the relationship between an application, several I/O families, and their plug-ins. Referring to FIG. 2, an application **201** requests services through one or more family FPIs, shown in FIG. 2 as File Manager API **202**, Block Storage API **203**, and SCSI Manager API **204**. The File Manager API **202**, Block Storage API **203**, and SCSI Manager API **204** are available to one or more applications in the user domain.

In one embodiment, the service requests from application **201** (and other applications) are sent through File Manager API **202**, Block Storage API **203**, and/or SCSI Manager API **204**, etc., and flow as messages to family FPI servers **205–207**, which reside in the kernel domain. In one embodiment, the messages are delivered using a kernel-supplied messaging service.

Any communication method may be used to communicate service requests to I/O families. In one embodiment, kernel messaging is used between the FPI libraries and the FPI server for a given family, between different families, and between plug-ins of one family and another family. The communication method used should be completely opaque to a client requesting a family service.

Each of the FPI servers **205–207** permit access to a distinct set of services. For example, File Manager FPI server **205** handles service for the file manager family of services. Similarly, the Block Storage FPI server **206** handles service requests for the block storage family of services.

Note that FIG. **2** shows three families linked by kernel messages. Messages flow from application level through a family to another family, and so on. For instance, a service

request may be communicated from application level to the file system family, resulting in one or more requests to the block storage family, and finally one or more to the SCSI family to complete a service request. Note that in one embodiment, there is no hierarchical relationship among families; all families are peers of each other.

Families in the Present Invention

A family provides a distinct set of services to the system. For example, one family may provide network services, while another provides access to a variety of block storage mediums. A family is associated with a set of devices that have similar characteristics, such as all display devices or all ADB devices.

In one embodiment, each family is implemented in software that runs in the computer system with applications. A family comprises software that includes a family programming interface and its associated FPI library or libraries for its clients, an FPI server, an activation model, a family expert, a plug-in programming interface for its plug-ins, and a family services library for its plug-ins.

FIG. **3** illustrates the interaction between these components. Referring to FIG. **3**, a family programming interface (FPI) **301** provides access to the family's services to one or more applications, such as application **302**. The FPI **301** also provides access to plug-ins from other families and to system software. That is, an FPI is designed to provide callers with services appropriate to a particular family, whether those calls originate from in the user domain or the operating system domain.

For example, when an application generates data for a video device, a display FPI tailored to the needs of video devices is used to gain access to display services. Likewise, when an application desires to input or output sound data, the application gains access to a sound family of services through an FPI. Therefore, the present invention provides family programming interfaces tailored to the needs of specific device families.

Service requests from application **302** (or other applications) are made through an FPI library **303**. In one embodiment, the FPI library **303** contains code that passes requests for service to the family FPI server **304**. In one embodiment, the FPI library **303** maps FPI function calls into messages (e.g., kernel messages) and sends them to the FPI server **304** of the family for servicing. In one embodiment, a family **305** may provide two versions of its FPI library **303**, one that runs in the user domain and one that runs in the operating system kernel domain.

In one embodiment, FPI server **304** runs in the kernel domain and responds to service requests from family clients (e.g., applications, other families, etc.). FPI server **304** responds to a request according to the activation model (not shown) of the family **305**. In one embodiment, the activation model comprises code that provides the runtime environment of the family and its plug-ins. For instance, FPI server **304** may put a request in a queue or may call a plug-in directly to service the request. As shown, the FPI server **304** forwards a request to the family **305** using a procedure call. Note that if FPI library **303** and the FPI server **304** use kernel messaging to communicate, the FPI server **304** provides a message port.

Each family **305** includes an expert (not shown) to maintain knowledge of the set of family devices. In one embodiment, the expert comprises code within a family **305** that maintains knowledge of the set of family plug-ins within the system. At system startup and each time a change occurs, the expert is notified.

In one embodiment, the expert may maintain the set of family services using a central device registry in the system.

The expert scans the device registry for plug-ins that belong to its family. For example, a display family expert looks for display device entries. When a family expert finds an entry for a family plug-in, it instantiates the plug-in, making it available to clients of the family. In one embodiment, the system notifies the family expert on an ongoing basis about new and deleted plug-ins in the device registry. As a result, the set of plug-ins known to and available through the family remains current with changes in system configuration.

Note that family experts do not add or alter information in the device registry nor do they scan hardware. In one embodiment, the present invention includes another level of families (i.e., low-level families) whose responsibility is to discover devices by scanning hardware and installing and removing information for the device registry. These low-level families are the same as the families previously discussed above (i.e., high level family) in other ways, i.e. they have experts, services, an FPI, a library, an activation model and plug-ins. The low-level families' clients are usually other families rather than applications. In one embodiment, families are insulated from knowledge of physical connectivity. Experts and the device registry are discussed in more detail below.

A plug-in programming interface (PPI) **306** provides a family-to-plug-in interface that defines the entry points a plug-in supports so that it can be called and a plug-in-to-family interface that defines the routines plug-ins call when certain events, such as an I/O completion, occur. In addition, PPI **306** defines the path through which the family and its plug-in exchange data.

A family services library (not shown) is a collection of routines that provide services to the plug-ins of a family. The services are specific to a given family and they may be layered on top of services provided by the kernel. Within a family, the methods by which data is communicated, memory is allocated, interrupts are registered and timing services are provided may be implemented in the family services library. Family services libraries may also maintain state information needed by a family to dispatch and manage requests.

For example, a display family services library provides routines that deal with vertical blanking (which is a concern of display devices). Likewise, SCSI device drivers manipulate command blocks, so the SCSI family services library contains routines that allow block manipulation. A family services library that provides commonly needed routines simplifies the development of that family's plug-ins.

Through the PPI **306**, a call is made to a plug-in **307**. In one embodiment, a plug-in, such as plug-in **307**, comprises dynamically loaded code that runs in the kernel's address space to provide an instance of the service provided by a family. For example, within the file systems family, a plug-in implements file-system-specific services. The plug-ins understand how data is formatted in a particular file system such as HFS or DOS-FAT. On the other hand, it is not the responsibility of file systems family plug-ins to obtain data from a physical device. In order to obtain data from a physical device, a file system family plug-in communicates to, for instance, a block storage family. In one embodiment, block storage plug-ins provide both media-specific drivers, such as a tape driver, a CD-ROM driver, or hard disk driver, and volume plug-ins that represent partitions on a given physical disk. Block storage plug-ins in turn may make SCSI family API calls to access data across the SCSI bus on a physical disk. Note that in the present invention, plug-ins are a superset of device drivers. For instance, plug-ins may include code that does not use hardware. For instance, file system and block storage plug-ins are not drivers (in that drivers back hardware).

Applications, plug-ins from other I/O families, and other system software can request the services provided by a family's plug-ins through the family's FPI. Note also that plug-ins are designed to operate in the environment set forth by their family activation model.

In one embodiment, a plug-in may comprises two code sections, a main code section that runs in a task in the kernel domain and an interrupt level code section that services hardware interrupts if the plug-in is, for instance, a device driver. In one embodiment, only work that cannot be done at task level in the main code section should be done at interrupt level. In one embodiment, all plug-ins have a main code section, but not all have interrupt level code sections.

The main code section executes and responds to client service requests made through the FPI. For example, sound family plug-ins respond to sound family specific requests such as sound playback mode setting (stereo, mono, sample size and rate), sound play requests, sound play cancellation, etc. The interrupt level code section executes and responds to interrupts from a physical device. In one embodiment, the interrupt level code section performs only essential functions, deferring all other work to a higher execution levels.

Also because all of the services associated with a particular family are tuned to the same needs and requirements, the drivers or plug-ins for a given family may be as simple as possible.

Family Programming Interfaces

In the present invention, a family provides either a user-mode or a kernel-mode FPI library, or both, to support the family's FPI. FIG. **4** illustrates one embodiment of the I/O architecture of the present invention. Referring to FIG. **4**, three instances of families **401–403** are shown operating in the kernel environment. Although three families are shown, the present invention may have any number of families.

In the user mode, two user-mode FPI libraries, $Xlib_u$ **404** and $Zlib_u$ **405**, are shown that support the FPIs for families X and Z, respectively. In the kernel environment, two kernel-mode FPI libraries, $Ylib_k$ **406** and $Zlib_k$, **407**, for families Y and Z, respectively, are shown.

Both the user-mode and the kernel-mode FPI libraries present the same FPI to clients. In other words, a single FPI is the only way family services can be accessed. In one embodiment, the user-mode and kernel mode libraries are not the same. This may occur when certain operations have meaning in one mode and not the other. For example, operations that are implemented in the user-mode library, such as copying data across address-space boundaries, may be unnecessary in the kernel library.

In response to service requests, FPI libraries **404** and **405** map FPI functions into messages for communication from the user mode to the kernel mode. In one embodiment, the messages are kernel messages.

The service requests from other families are generated by plug-ins that make calls on libraries, such as FPI libraries **406** and **407**. In one embodiment, FPI libraries **406** and **407** map FPI functions into kernel messages and communicate those messages to FPI servers such as Y FPI server **409** and Z FPI server **410** respectively. Other embodiments may use mechanisms other than kernel messaging to communicate information.

In the example, the Z family **403** has both a user-mode library **405** and a kernel-mode library **407**. Therefore, the services of the Z family may be accessed from both the user mode and the kernel mode.

In response to service request messages, X FPI server **408**, Y FPI server **409** and Z FPI server **410** dispatch requests for services to their families. In one embodiment, each of FPI servers **408–410** receives a kernel message, maps the message into a FPI function called by the client, and then calls the function in the family implementation (**414–416**).

In one embodiment, there is a one-to-one correspondence between the FPI functions called by clients and the function called by FPI servers **408–410** as a result. The calls from FPI serves **408–410** are transferred via interfaces **411–413**. For instance, X interface **411** represents the interface presented to the FPI server **408** by the X family **414**. It is exactly the same as the FPI available to applications or other system software. The same is true of Y interface **412** and Z interface **413**.

The X family implementation **414** represents the family activation model that defines how requests communicated from server **408** are serviced by the family and plug-in(s). In one embodiment, X family implementation **414** comprises family code interfacing to plug-in code that completes the service requests from application **400** via server **408**. Similarly, the Y family implementation **415** and Z family implementation **416** define their family's plug-in activation models.

X plug-in **417**, Y plug-in **418** and Z plug-in **419** operate within the activation model mandated by the family and provide code and data exports. The required code and data exports and the activation model for each family of drivers is family specific and different.

Extending Family Programming Interfaces

A plug-in may provide a plug-in-specific interface that extends its functionality beyond that provided by its family. This is useful in a number of situations. For example, a block storage plug-in for a CD-ROM device may provide a block storage plug-in interface required of the CD-ROM device as well as an interface that allows knowledgeable application software to control audio volume and to play, pause, stop, and so forth. Such added capabilities require a plug-in-specific API.

If a device wishes to export extended functionality outside the family framework, a separate message port is provided by the device and an interface library for that portion of the device driver. FIG. **5** illustrates the extension of a family programming interface.

Referring to FIG. **5**, a plug-in module, Z plug-in **501**, extends beyond the Z family boundary to interface to family implementation D **502** as well. A plug-in that has an extended API offers features in addition to those available to clients through it's family's FPI. In order to provide extra services, the plug-in provides additional software shown in FIG. **5** as an interface library Dlib$_u$ **503**, the message port code D FPI server **504**, and the code that implements the extra features D **505**.

Sharing Code and Data Between Plug-ins

In one embodiment, two or more plug-ins can share data or code or both, regardless of whether the plug-ins belong to the same family or to different families. Sharing code or data is desirable when a single device is controlled by two or more families. Such a device needs a plug-in for each family. These plug-ins can share libraries that contain information about the device state and common code. FIG. **6** illustrates two plug-ins that belong to separate families and that share code and data.

Plug-ins can share code and data through shared libraries. Using shared libraries for plug-ins that share code or data allows the plug-ins to be instantiated independently without encountering problems related to simultaneous instantiation. Referring to FIG. **6**, the first plug-in **601** to be opened and initialized obtains access to the shared libraries. At this point, the first plug-in **601** does not share access. When the second plug-in **602** is opened and initialized, a new connection to the shared libraries is created. From that point, the two plug-ins contend with each other for access to the shared libraries.

Sharing code or data may also be desirable in certain special cases. For instance, two or more separate device drivers may share data as a way to arbitrate access to a shared device. An example of this is a single device that provides network capabilities and real time clock. Each of these functions belong to a distinct family but may originate in a single physical device.

Activation Models in the Present Invention

An activation model defines how the family is implemented and the environment within which plug-ins of the family execute. In one embodiment, the activation model of the family defines the tasking model a family uses, the opportunities the family plug-ins have to execute and the context of those opportunities (for instance, are the plug-ins called at task time, during privileged mode interrupt handling, and so forth), the knowledge about states and processes that a family and its plug-ins are expected to have, and the portion of the service requested by the client that is performed by the family and the portion that is performed by the plug-ins.

Each model provides a distinctly different environment for the plug-ins to the family, and different implementation options for the family software. Examples of activation models include the single-task model, the task-per-plug-in model, and the task-per-request model. Each is described in further detail below. Note that although three activation models are discussed, the choice of activation model is a design choice and different models may be used based on the needs and requirements of the family.

In one embodiment, the activation model uses kernel messaging as the interface between the FPI libraries that family clients link to and the FPI servers in order to provide the asynchronous or synchronous behavior desired by the family client. Within the activation model, asynchronous I/O requests are provided with a task context. In all cases, the implementation of the FPI server depends on the family activation model.

The choice of activation model limits the plug-in implementation choices. For example, the activation model defines the interaction between a driver's hardware interrupt level and the family environment in which the main driver runs. Therefore, plug-ins conform to the activation model employed by its family.

Single-Task Model

One of the activation models that may be employed by a family is referred to herein as the single-task activation model. In the single-task activation model, the family runs as a single monolithic task which is fed from a request queue and from interrupts delivered by plug-ins. Requests are delivered from the FPI library to an accept function that enqueues the request for processing by the family's processing task and wakes the task if it is sleeping. Queuing, synchronization, and communication mechanism within the family follow a set of rules specified by the family.

The interface between the FPI Server and a family implementation using the single-task model is asynchronous. Regardless of whether the family client called a function synchronously or asynchronously, the FPI server calls the family code asynchronously. The FPI server maintains a set

of kernel message IDs that correspond to messages to which the FPI server has not yet replied. The concept of maintaining kernel message IDs corresponding to pending I/O server request messages is well-known in the art;

Consider as an example family **700**, which uses the single-task activation model, shown in FIG. **7**. Referring to FIG. **7**, an application **710** is shown generating a service request to the family's APIs **711**. APIs **711** contain at least one library in which service requests are mapped to FP1 functions. The FPI functions are forwarded to the family's FPI server **701**. FPI server **701** dispatches the FPI function to family implementation **703**, which includes various protocols and a network device driver that operate as a single task. Each protocol layer provides a different level of service.

The FPI server **701** is an accept function that executes in response to the calling client via the FPI library (not shown). An accept function, unlike a message-receive-based kernel task, is able to access data within the user and kernel bands directly. The accept function messaging model requires that FPI server **701** be re-entrant because the calling client task may be preempted by another client task making service requests.

When an I/O request completes within the family's environment, a completion notification is sent back to the FPI server **701**, which converts the completion notification into the appropriate kernel message ID reply. The kernel message ID reply is then forwarded to the application that generated the service request.

With a single-task model, the family implementation is insulated from the kernel in that the implementation does it not have kernel structures, IDs, or tasking knowledge. On the other hand, the relationship between FPI server **701** and family code **702** is asynchronous, and has internal knowledge of data structures and communication mechanisms of the family.

The single-task model may be advantageously employed for families of devices that have one of several characteristics: (1) each I/O request requires little effort of the processing unit. This applies not only to keyboard or mouse devices but also to DMA devices to the extent that the processing unit need only set up the transfer, (2) no more than one I/O request is handled at once, such that, for instance, the family does not allow interleaving of I/O requests. This might apply to sound, for example, or to any device for which exclusive reservation is required (i.e., where only one client can use a device at a time). The opposite of a shared resource. Little effort for the processor exists where the processor initiates an I/O request and then is not involved until the request completes, or (3) the family to be implemented provides its own scheduling mechanisms independent of the underlying kernel scheduling. This applies to the Unix™ stream programming model.

Task-Per-Plug-In Model

For each plug-in instantiated by the family, the family creates a task that provides the context within which the plug-in operates.

FIG. **8** illustrates the task-per-plug-in model. Referring to FIG. **8**, an application **801** generates service requests for the family, which are sent to FPI **802**. Using an FPI library, the FPI **802** generates a kernel message according to the family activation model **804** and a driver, such as plug-in driver **805**.

In one embodiment, the FPI server **803** is a simple task-based message-receive loop or an accept function. FPI server **803** receives requests from calling clients and passes those requests to the family code **804**. The FPI server **803** is responsible for making the data associated with a request available to the family, which in turn makes it available to the plug-in that services the request. In some instances, this responsibility includes copying or mapping buffers associated with the original request message to move the data from user address space to the kernel level area.

The family code **804** consists in part of one or more tasks, one for each family plug-in. The tasks act as a wrapper for the family plug-ins such that all tasking knowledge is located in the family code. A wrapper is a piece of code that insulates called code from the original calling code. The wrapper provides services to the called code that the called code is not aware of.

When a plug-in's task receives a service request (by whatever mechanisms the family implementation uses), the task calls its plug-in's entry points, waits for the plug-in's response, and then responds to the service request.

The plug-in performs the work to actually service the request. Each plug-in does not need to know about the tasking model used by the family or how to respond to event queues and other family mechanisms; it only needs to know how to perform its particular function.

For concurrent drivers, all queuing and state information describing an I/O request is contained within the plug-in code and data and within any queued requests. The FPI library forwards all requests regardless of the status of outstanding I/O requests to the plug-in. When the client makes a synchronous service request, the FPI library sends a synchronous kernel message. This blocks the requesting client, but the plug-in's task continues to run within its own task context. This permits clients to make requests of this plug-in even while another client's synchronous request is being processed.

In some cases of a family, a driver (e.g., **805**) can be either concurrent or nonconcurrent. Nevertheless, clients of the family may make synchronous and asynchronous requests, even though the nonconcurrent drivers can handle only one request at a time. The device manager FPI server **803** knows that concurrent drivers cannot handle multiple requests concurrently. Therefore, FPI server **803** provides a mechanism to queue client requests and makes no subsequent requests to a task until the task signals completion of an earlier I/O request.

When a client calls a family function asynchronously, the FPI library sends an asynchronous kernel message to the FPI server and returns to the caller. When a client calls a family function synchronously, the FPI library sends a synchronous kernel message to the FPI server and does not return to the caller until the FPI server replies to the message, thus blocking the caller's execution until the I/O request is complete.

In either case, the behaviors of the device manager FPI server **803** is exactly the same: for all incoming requests, it either queues the request or passes it to the family task, depending on whether the target plug-in is busy. When the plug-in signals that the I/O operation is complete, the FPI server **803** replies to the kernel message. When the FPI library receives the reply, it either returns to the synchronous client, unblocking its execution or it notifies the asynchronous client about the I/O completion.

The task-per-plug-in model is intermediate between the single-task and task-per-request models in terms of the number of tasks it typically uses. The task-per-plug-in model is advantageously used where the processing of I/O requests varies widely among the plug-ins.

Task-Per-Request Model

The task-per-request model shares the following characteristics with the two activation models already discussed:

(1) the FPI library to FPI server communication provides the synchronous or asynchronous calling behavior requested by family clients, and (2) the FPI library and FPI server use kernel messages to communicate I/O requests between themselves. However, in the task-per-request model, the FPI server's interface to the family implementation is completely synchronous.

In one embodiment, one or more internal family request server tasks, and, optionally, an accept function, wait for messages on the family message port. An arriving message containing information describing an I/O request awakens one of the request server tasks, which calls a family function to service the request. All state information necessary to handle the request is maintained in local variables. The request server task is blocked until the I/O request completes, at which time it replies to the kernel message from the FPI library to indicate the result of the operation. After replying, the request server task waits for more messages from the FPI library.

As a consequence of the synchronous nature of the interface between the FPI server and the family implementation, code calling through this interface remains running as a blockable task. This calling code is either the request server task provided by the family to service the I/O (for asynchronous I/O requests) or the task of the requester of the I/O (for certain optimized synchronous requests).

The task-per-request model is advantageously employed for a family where an I/O request can require continuous attention from the processor and multiple I/O requests can be in progress simultaneously. A family that supports dumb, high bandwidth devices is a good candidate for this model. In one embodiment, the file manager family uses the task-per-request model. This programming model requires the family plug-in code to have tasking knowledge and to use kernel facilities to synchronize multiple threads of execution contending for family and system resources.

Unless there are multiple task switches within a family, the tasking overhead is identical within all of the activation models. The shortest task path from application to I/O is completely synchronous because all code runs on the caller's task thread.

Providing at least one level of asynchronous call between an application and an I/O request results in better latency results from the user perspective. Within the file system, a task switch at a file manager API level allows a user-visible application, such as the Finder™, to continue. The file manager creates an I/O tasks to handle the I/O request, and that task is used via synchronous calls by the block storage and SCSI families to complete their part in I/O transaction processing.

The Device Registry of the Present Invention

The device registry of the present invention comprises an operating system naming service that stores system information. In one embodiment, the device registry is responsible for driver replacement and overloading capability so that drivers may be updated, as well as for supporting dynamic driver loading and unloading.

In one embodiment, the device registry of the present invention is a tree-structured collection of entries, each of which can contain an arbitrary number of name-value pairs called properties. Family experts examine the device registry to locate devices or plug-ins available to the family. Low-level experts, discussed below, describe platform hardware by populating the device registry with device nodes for insertion of devices that will be available for use by applications.

In one embodiment, the device registry contains a device subtree pertinent to the I/O architecture of the present invention. The device tree describes the configuration and connectivity of the hardware in the system. Each entry in the device tree has properties that describe the hardware represented by the entry and that contain a reference to the driver in control of the device.

Multiple low-level experts are used, where each such expert is aware of the connection scheme of physical devices to the system and installs and removes that information in the device tree portion of the device registry. For example a low-level expert, referred to herein as a bus expert or a motherboard expert, has specific knowledge of a piece of hardware such as a bus or a motherboard. Also, a SCSI bus expert scans a SCSI bus for devices, and installs an entry into the device tree for each device that it finds. The SCSI bus expert knows nothing about a particular device for which it installs an entry. As part of the installation, a driver gets associated with the entry by the SCSI bus expert. The driver knows the capabilities of the device and specifies that the device belongs to a given family. This information is provided as part of the driver or plug-in descriptive structure required of all plug-ins as part of their PPI implementation.

Low-level experts and family experts use a device registry notification mechanism to recognize changes in the system configuration and to take family-specific action in response to those changes.

An example of how family experts, low-level experts, and the device registry service operate together to stay aware of dynamic changes in system configuration follows: Suppose a motherboard expert notices that a new bus, a new network interface and new video device have appeared within the system. The motherboard expert adds a bus node, a network node, and a video node to the device tree portion of the device registry. The device registry service notifies all software that registered to receive notifications of these events.

Once notified that changes have occurred in the device registry, the networking and video family experts scan the device registry and notice the new entry belonging to their family type. Each of the experts adds an entry in the family subtree portion of the device registry.

The SCSI bus expert notices an additional bus, and probes for SCSI devices. It adds a node to the device registry for each SCSI device that it finds. New SCSI devices in the device registry result in perusal of the device registry by the block storage family expert. The block storage expert notices the new SCSI devices and loads the appropriate drivers, and creates the appropriate device registry entries, to make these volumes available to the file manager. The file manager receives notification of changes to the block storage family portion of the device registry, and notifies the Finder™ that volumes are available. These volumes then appear on the user's desktop.

Whereas, many alterations and modifications of the present invention will no doubt become apparent to a person of ordinary skill in the art after having read the foregoing description, it is to be understood that the particular embodiment shown and described by way of illustration are in no way to be considered limiting. Therefore, reference to the details of the various embodiments are not intended to limit the scope of the claims which themselves recite only those features regarded as essential to the invention.

Thus, a method and apparatus for handling I/O requests in a computer system has been described.

We claim:

1. A computer system comprising:

a bus;

at least one memory coupled to the bus for storing data and programming instructions that include applications and an operating system; and

a processing unit coupled to the bus and running the operating system and applications by executing programming instructions, wherein an application has a first plurality of tailored distinct programming inter-

faces available to access a plurality of separate sets of computer system services provided through the operating system of the computer system via service requests.

2. The computer system defined in claim 3 wherein each of the first plurality of tailored distinct programming interfaces are tailored to a type of I/O service provided by each set of I/O services.

3. A computer system comprising:

a bus;

at least one memory coupled to the bus for storing data and programming instructions that include applications and an operating system, wherein the operating system comprises a plurality of servers, and each of the first plurality of programming interfaces transfer service requests to one of the plurality of servers, wherein each of the plurality of servers responds to service requests from clients of the separate sets of I/O services; and

a processing unit coupled to the bus and running the operating system and applications by executing programing instructions, wherein an application has a first plurality of tailored distinct programming interfaces available to access a plurality of separate sets of I/O services provided through the operating system via service requests.

4. The computer system defined in claim 3 wherein service requests are transferred as messages in a messaging system.

5. The computer system defined in claim 4 wherein each of the plurality of servers supports a message port.

6. The computer system defined in claim 3 wherein at least one of the plurality of servers is responsive to service requests from applications and from at least one other set of I/O services.

7. The computer system defined in claim 3 wherein the operating system further comprises a plurality of activation models, wherein each of the plurality of activation models is associated with one of the plurality of servers to provide a runtime environment for the set of I/O services to which access is provided by said one of the plurality of servers.

8. The computer system defined in claim 7 wherein at least one instance of a service is called by one of the plurality of servers for execution in an environment set forth by one of the plurality of activation models.

9. A computer system comprising:

a bus;

at least one memory coupled to the bus for storing data and programming instructions that comprise applications and an operating system;

a processing unit coupled to the bus and running the operating system and applications by executing programming instructions, wherein the operating system provides computer system services through a tailored distinct one of a plurality of program structures, each tailored distinct program structure comprising:

a first programming interface for receiving service requests for a set of computer system I/O services of a first type,

a first server coupled to receive service requests and to dispatch service requests to the computer system I/O services,

an activation model to define an operating environment in which a service request is to be serviced by the set of computer system I/O services, and

at least one specific instance of the set of computer system I/O services that operate within the activation model.

10. The computer system defined in claim 9 wherein the first programming interface is responsive to request from applications and from other program structures.

11. The computer system defined in claim 9 wherein the first programming interface comprises at least one library for converting functions into messages.

12. The computer system defined in claim 9 wherein the first server receives a message corresponding a service request from the first programming interface, maps the message into a function called by the client, and then calls the function.

13. The computer system defined in claim 9 wherein the message comprises a kernel message.

14. A computer system comprising:

a bus;

at least one memory coupled to the bus for storing data and programming instructions that comprise applications and an operating system;

a processing unit coupled to the bus and running the operating system and applications by executing programming instructions, wherein the operation system provides input/output (I/O) services through a tailored distinct one of plurality of program structures, each tailored distinct program structure comprising:

a first programming interface for receiving service requests for a set of I/O services of a first type,

a first server coupled to receive service requests and to dispatch service requests to the I/O services,

an activation model to define operating environment in which a service request is to be serviced by the set of I/O services, and

at least one specific instance of the set of I/0 services that operate within the activation model, wherein one of the said at least one specific instances comprises a service that accesses another program structure, and further wherein said one of said at least one specific instances communicates to said another program structure of a second type using a message created using a library sent to the server of said another program structure.

15. The computer system defined in claim 9 wherein two or more I/O services share code or data.

16. The computer system defined in claim 15 wherein said two or more I/O services are different types.

17. The computer system defined in claim 9 wherein the program structure further comprises a storage mechanism to maintain identification of available services to which access is provided via the first server.

18. A computer implemented method of accessing I/O services of a first type, said computer implemented method comprising the steps of:

generating a service request for a first type of I/O services;

a tailored distinct family server, operating in an operating system environment and dedicated to providing access to service requests for the first type of I/O service, receiving and responding to the service request based on an activation model specific to the first type of I/O services; and

a processor running an instance of the first type of I/O services that is interfaces to the file server to satisfy the service request.

19. The method defined in claim 18 wherein the service request is generated by an application.

20. The method defined in claim 18 wherein the service request is generated by an instance of an I/O service running in the operating system environment.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 5,915,131
DATED : June 22, 1999
INVENTOR(S) : Knight, et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

In column 15 at line 14 delete "the" and insert -- a --
In column 15 at line 54 delete ":" and insert -- ; --
In column 16 at line 20 delete "operation" and insert -- operating --
In column 16 at line 58 delete "interfaces" and insert -- interfaced --

Signed and Sealed this

Eighteenth Day of January, 2000

Attest:

Q. TODD DICKINSON

Attesting Officer

Commissioner of Patents and Trademarks